A hands-on introduction to end-to-end data projects in Python for SAS Programmers

A few notes before we get started...

Google Colab Setup

- To save a copy of this notebook, along with any notes/edits you make, use the File menu: **File** -> **Save a copy in Drive**
- To enable line numbers, use the Tools menu: **Tools** -> **Settings** -> **Editor** -> **Show line numbers** -> **Save**
- To disable Gemini, use the Tools menu: **Tools** -> **Settings** -> **AI Assistance** -> **Hide generative AI features** -> **Close**
- To enable the Table of Contents, use the View menu: **View** -> **Table of contents**

Google Accounts

- If you see a **Sign In** button in the upper right corner, we recommend signing into a Google account. This will enable you to make a copy of this notebook (e.g., to take notes during the workshop), as well as to execute example code.
- If you don't already have a Google account, you can create one for free at https://accounts.google.com/signup

Looking for Extra Credit?

- Please let us know if you spot any typos!

Section 0. Setup and Definitions

Install and import packages used throughout this Notebook

Instructions: Click anywhere in the code cell below, and run the cell by pressing Shift-Enter or by clicking the triangular "play" icon in the upper left corner of the cell.

In [1]:
```
# Install the great_tables and parquet-tools packages, which will be used to create output later
!pip install great-tables parquet-tools

# Import from standard-library packages
import pathlib

# Import from third-party packages
from great_tables import GT, style, loc
from IPython.display import display
import pandas
from statsmodels.formula.api import logit
```

```
Collecting great-tables
  Downloading great_tables-0.11.0-py3-none-any.whl.metadata (10 kB)
Collecting parquet-tools
  Downloading parquet_tools-0.2.16-py3-none-any.whl.metadata (3.8 kB)
Collecting commonmark>=0.9.1 (from great-tables)
  Downloading commonmark-0.9.1-py2.py3-none-any.whl.metadata (5.7 kB)
Collecting htmltools>=0.4.1 (from great-tables)
  Downloading htmltools-0.5.3-py3-none-any.whl.metadata (3.3 kB)
Requirement already satisfied: importlib-metadata in /usr/local/lib/python3.10/dist-packages (from great
-tables) (8.4.0)
Requirement already satisfied: typing-extensions>=3.10.0.0 in /usr/local/lib/python3.10/dist-packages (f
rom great-tables) (4.12.2)
Requirement already satisfied: numpy>=1.22.4 in /usr/local/lib/python3.10/dist-packages (from great-tabl
es) (1.26.4)
Requirement already satisfied: Babel>=2.13.1 in /usr/local/lib/python3.10/dist-packages (from great-tabl
```

```
es) (2.16.0)
Requirement already satisfied: importlib-resources in /usr/local/lib/python3.10/dist-packages (from grea
t-tables) (6.4.4)
Collecting boto3<2.0.0,>=1.34.11 (from parquet-tools)
  Downloading boto3-1.35.12-py3-none-any.whl.metadata (6.6 kB)
Collecting colorama<0.5.0,>=0.4.6 (from parquet-tools)
  Downloading colorama-0.4.6-py2.py3-none-any.whl.metadata (17 kB)
Collecting halo<0.0.32,>=0.0.31 (from parquet-tools)
  Downloading halo-0.0.31.tar.gz (11 kB)
  Preparing metadata (setup.py) ... done
Requirement already satisfied: pandas<3.0.0,>=2.1.4 in /usr/local/lib/python3.10/dist-packages (from par
quet-tools) (2.1.4)
Requirement already satisfied: pyarrow in /usr/local/lib/python3.10/dist-packages (from parquet-tools) (
14.0.2)
Requirement already satisfied: tabulate<0.10.0,>=0.9.0 in /usr/local/lib/python3.10/dist-packages (from
parquet-tools) (0.9.0)
Collecting thrift<0.17.0,>=0.16.0 (from parquet-tools)
  Downloading thrift-0.16.0.tar.gz (59 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 59.6/59.6 kB 1.1 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Collecting botocore<1.36.0,>=1.35.12 (from boto3<2.0.0,>=1.34.11->parquet-tools)
  Downloading botocore-1.35.12-py3-none-any.whl.metadata (5.7 kB)
Collecting jmespath<2.0.0,>=0.7.1 (from boto3<2.0.0,>=1.34.11->parquet-tools)
  Downloading jmespath-1.0.1-py3-none-any.whl.metadata (7.6 kB)
Collecting s3transfer<0.11.0,>=0.10.0 (from boto3<2.0.0,>=1.34.11->parquet-tools)
  Downloading s3transfer-0.10.2-py3-none-any.whl.metadata (1.7 kB)
Collecting log_symbols>=0.0.14 (from halo<0.0.32,>=0.0.31->parquet-tools)
  Downloading log_symbols-0.0.14-py3-none-any.whl.metadata (523 bytes)
Collecting spinners>=0.0.24 (from halo<0.0.32,>=0.0.31->parquet-tools)
  Downloading spinners-0.0.24-py3-none-any.whl.metadata (576 bytes)
Requirement already satisfied: termcolor>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from halo<0.
0.32,>=0.0.31->parquet-tools) (2.4.0)
Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.10/dist-packages (from halo<0.0.32,
>=0.0.31->parquet-tools) (1.16.0)
Requirement already satisfied: packaging>=20.9 in /usr/local/lib/python3.10/dist-packages (from htmltool
s>=0.4.1->great-tables) (24.1)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10/dist-packages (from p
andas<3.0.0,>=2.1.4->parquet-tools) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas<3.0.
```

```
0,>=2.1.4->parquet-tools) (2024.1)
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dist-packages (from pandas<3.
0.0,>=2.1.4->parquet-tools) (2024.1)
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.10/dist-packages (from importlib-meta
data->great-tables) (3.20.1)
Requirement already satisfied: urllib3!=2.2.0,<3,>=1.25.4 in /usr/local/lib/python3.10/dist-packages (fr
om botocore<1.36.0,>=1.35.12->boto3<2.0.0,>=1.34.11->parquet-tools) (2.0.7)
Downloading great_tables-0.11.0-py3-none-any.whl (1.3 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.3/1.3 MB 9.9 MB/s eta 0:00:00
Downloading parquet_tools-0.2.16-py3-none-any.whl (31 kB)
Downloading boto3-1.35.12-py3-none-any.whl (139 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 139.2/139.2 kB 7.8 MB/s eta 0:00:00
Downloading colorama-0.4.6-py2.py3-none-any.whl (25 kB)
Downloading commonmark-0.9.1-py2.py3-none-any.whl (51 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 51.1/51.1 kB 2.6 MB/s eta 0:00:00
Downloading htmltools-0.5.3-py3-none-any.whl (83 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 83.2/83.2 kB 5.1 MB/s eta 0:00:00
Downloading botocore-1.35.12-py3-none-any.whl (12.5 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 12.5/12.5 MB 42.8 MB/s eta 0:00:00
Downloading jmespath-1.0.1-py3-none-any.whl (20 kB)
Downloading log_symbols-0.0.14-py3-none-any.whl (3.1 kB)
Downloading s3transfer-0.10.2-py3-none-any.whl (82 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 82.7/82.7 kB 4.9 MB/s eta 0:00:00
Downloading spinners-0.0.24-py3-none-any.whl (5.5 kB)
Building wheels for collected packages: halo, thrift
  Building wheel for halo (setup.py) ... done
  Created wheel for halo: filename=halo-0.0.31-py3-none-any.whl size=11233 sha256=66b41d5cc1ef5d44e59175
b42e7254f593e2118f2453206c554027937ce03423
  Stored in directory: /root/.cache/pip/wheels/5a/d9/8a/b4f14c44aba7c164d4379eca6f1dde59360050406b1edaec
24
  Building wheel for thrift (setup.py) ... done
  Created wheel for thrift: filename=thrift-0.16.0-cp310-cp310-linux_x86_64.whl size=373866 sha256=0079d
918795a6358b2f25858082e28bb0ed33672b1d75d04690964a0d6381bd2
  Stored in directory: /root/.cache/pip/wheels/52/f8/d2/acfd995e8247eb0cad372fa6a640a5fcf279ab2ed7c5c449
0e
Successfully built halo thrift
Installing collected packages: spinners, commonmark, thrift, jmespath, htmltools, colorama, log_symbols,
great-tables, botocore, s3transfer, halo, boto3, parquet-tools
Successfully installed boto3-1.35.12 botocore-1.35.12 colorama-0.4.6 commonmark-0.9.1 great-tables-0.11.
```

```
0 halo-0.0.31 htmltools-0.5.3 jmespath-1.0.1 log_symbols-0.0.14 parquet-tools-0.2.16 s3transfer-0.10.2 s
pinners-0.0.24 thrift-0.16.0
```

Notes:

1. The `!` operator indicates a shell command to the operating system, which allows us to install packages not included in Colab. Specifically, we install the aptly named `great-tables` for building a summary report, as well as the `parquet-tools` package for interacting with external file in a parquet format.
2. These packages are installed using the standard `pip` utility, where "pip" is sometimes described as a recursive acronym meaning "pip installs packages."
3. By default, Python only loads a small portion of available functionality, so we also need to `import` several things to make them available in our current session.
4. The `pathlib` package is used to interact with the file system in a consistent way across platforms. This package is part of the Python standard library, meaning it should be part of any Python installation.
5. Python comes with a large standard library because of its "batteries included" philosophy, and numerous third-party modules are also actively developed and made freely available through sites like https://github.com/ and https://pypi.org/
6. In particular, `IPython`, `pandas`, and `statsmodels` are third-party packages already installed in Colab:
    - The `display` method from `IPython` will be used to display data throughout this notebook. This is the Python equivalent of something like PROC PRINT or a `select` clause in PROC SQL in SAS. (Note: When executing Python code outside of a notebook, we would typically use the builtin Python `print` function instead.)
    - The `pandas` package provides the `DataFrame` object. Like their R counterpart, DataFrames are two-dimensional arrays of values comparable to SAS datasets.
    - The `logit` method of the `statsmodels` package will be used to build a logistic regression model. This is the Python equivalent of the something like PROC LOGISTIC in SAS.
7. In case you're wondering "IPython" stands for "Interactive Python." Google Colab is built on top of JupyterLab, and

JupyterLab is built on top of IPython, which is why IPython is already available inside Colab.

8. Similarly, since Colab is intended to be a sandbox for data analysis, popular packages like `pandas` and `statsmodels` are also available by default.

Define constants and helper functions to be used throughout this Notebook

Instructions: Click anywhere in the code cell below, and run the cell by pressing Shift-Enter or by clicking the triangular "play" icon in the upper left corner of the cell.

In [2]:
```python
# Set a lower bound to use when checking for valid date of birth (dob) values
dob_lower_bound = pandas.Timestamp(1900, 1, 1)

# Create a new directory for file output
output_dir = pathlib.Path.cwd() / 'permanent_datasets'
output_dir.mkdir(parents=True, exist_ok=True)

# Define three functions to use when checking data integrity
def confirm_unique_id_column(df: pandas.DataFrame, column_name: str) -> bool:
    """
        Returns a boolean indicating whether a column in a DataFrame can be
        treated as a unique id by checking that the number of unique, non-null
        values is the same as the number of rows in the DataFrame
    """
    return df[column_name].nunique() == len(df[column_name])

def confirm_foreign_key_column(df1: pandas.DataFrame, df2: pandas.DataFrame, col: str) -> bool:
    """
        Returns a boolean indicating whether column col in DataFrame df1 can be
        treated as a foreign key linking df1 to df2 by checking that all values
        are non-null and appear in DataFrame df2
    """
    return (
        # Compare the number of non-null rows is the same as the number of rows
        df1[col].count() == len(df1[col]) and
```

```
        # Confirm all values of column appear in the reference DataFrame
        all(df1[col].isin(df2[col]))
    )

def calculate_age(df: pandas.DataFrame, col1: str, col2: str) -> pandas.Series:
    """
    Returns a pandas Series object comprising the pairwise differences in years
    (rounded down) between date columns col1 and col2, with col1 assumed to be
    farther in the future than col2
    """
    return (
      (df[col1].dt.year - df[col2].dt.year) -
      ((df[col1].dt.month*100 + df[col1].dt.day) - (df[col2].dt.month*100 + df[col2].dt.day) < 0)
    )
```

Notes:

1. The `pandas.Timestamp` method is used to define an object representing the date January 1st, 1900. Because no units of time smaller than a day have been specified, the time component of this object will default to zero (midnight). This is analogous to using the SAS functions `mdy` and `dhms` to define SAS date and datetime variables, respectively.

2. Code like `pandas.Timestamp` uses object-oriented dot-notation to have the `pandas` object module invoke the sub-module object `Timestamp` nested inside of it. (Think Russian nesting dolls or turduckens.)

3. A directory named "permanent_datasets" is created in the current working directory (`cwd`) using a common combination of methods from the `pathlib` package. This directory will be used later to store copies of an analytic dataset on disk. Note, in particular, the use of a forward slash `/` to connect path components, which is borrowed from the way paths are delimited in Linux or Unix-like operating systems.

4. Three functions have been defined for later use as part of routine checks for data quality:

    - The `confirm_unique_id_column` function checks if the number of unique, non-missing values in a DataFrame column is the same as the length of that column.
    - The `confirm_foreign_key_column` function checks for missing values in a column in a DataFrame and then

validates the list of values against the corresponding column in a reference DataFrame.
    - The `calculate_age` function creates an age column based on the pairwise difference between two date columns in a DataFrame.
5. In Python, a function definition starts with the `def` keyword and ends with a `return` statement, allowing code to be parameterized and reused similar to a SAS macro.

6. The indentation inside the function definition is mandatory and indicates scope. The end of the indented block of statements signals the end of the function definition in Python, just as a `%mend` statement ends a SAS macro definition.

7. When brackets `[]` are used directly after the name of an object in Python, it generally means we are taking a subset of that object. For example, in the `calculate_age` function definition, brackets are used to select a single column out of a DataFrame.

8. Note that a single `=` is used on line 2 to assign a value to an object, while a double `==` is used for comparison on lines 15 and 25. This distinction does not exist in SAS.

9. Similarly, code like `PANDAS.TIMESTAMP` would likely produce an error because capitalization matters in Python, unlike in SAS.

Section 1. Data Access

As a first step, we need to bring our input data into Python. Sometimes, this might involve reading from an external database server, Excel file, or any number of other data sources. For this example, we'll assume everything has been provided in three CSV files:

- A dataset of patients having characteristics that could potentially included them in a study cohort.
- A dataset of medication dispenses, with each row representing a patient having been dispensed a specific amount of either Drug A or Drug B on a specific date.
- A dataset of diagnoses (dx) of heart failure (hf), with each row representing a patient having been diagnosed on a specific date with a specific heart-failure condition identified by an ICD-10 CM code.

Example 1.1 Import Data from GitHub

Type (or copy/paste) the following into the code cell immediately below, and then run that cell using Shift-Enter:

```
patients_df = pandas.read_csv(
    'https://raw.githubusercontent.com/saspy-bffs/wuss-2024-python-
how/main/input_data/fakes/patients-2024_08_04T13_09_15.csv',
    na_filter=False,
)
dispenses_df = pandas.read_csv(
    'https://raw.githubusercontent.com/saspy-bffs/wuss-2024-python-
how/main/input_data/fakes/dispenses-2024_08_04T13_09_15.csv',
    na_filter=False,
)
diagnoses_df = pandas.read_csv(
    'https://raw.githubusercontent.com/saspy-bffs/wuss-2024-python-
how/main/input_data/fakes/diagnoses-2024_08_04T13_09_15.csv',
    na_filter=False,
)
```

In [3]:
```
patients_df = pandas.read_csv(
    'https://raw.githubusercontent.com/saspy-bffs/wuss-2024-python-how/main/input_data/fakes/patients-202
    na_filter=False,
)
dispenses_df = pandas.read_csv(
    'https://raw.githubusercontent.com/saspy-bffs/wuss-2024-python-how/main/input_data/fakes/dispenses-20
    na_filter=False,
)
diagnoses_df = pandas.read_csv(
    'https://raw.githubusercontent.com/saspy-bffs/wuss-2024-python-how/main/input_data/fakes/diagnoses-20
    na_filter=False,
)
```

Notes about Example 1.1.

1. Three DataFrames are created using the pandas `read_csv` method.
2. The parameter `na_filter=False` sets missing values to `''` (an empty string), rather than `NaN` (Not a Number). Because `pandas` was originally built on a package called `numpy`, pandas defaults to treating all values like numbers.
3. In SAS, this import would typically be achieved using something like PROC IMPORT. (However, while PROC IMPORT always prints to the SAS log, `read_csv` is silent unless an error has occurred.) For example, the following would import the patients dataset into SAS:

```
FILENAME indata URL "https://raw.githubusercontent.com/saspy-bffs/wuss-2024-python-
how/main/input_data/fakes/patients-2024_08_04T13_09_15.csv";
PROC IMPORT DATAFILE=indata
            OUT=patients_ds
            DBMS=csv
            REPLACE;
    GETNAMES=yes;
    GUESSINGROWS=1000;
RUN;
```

1. You may find calling the `read_csv` method three times repetitive. For extra credit, try executing the code below to

loop over the list of input datasets:

```
# Create an empty dictionary object
df_dict = {}
# Loop over the input datasets, import them, and store each DataFrame in the dictionary
for name in ['patients', 'dispenses', 'diagnoses']:
    df_dict[name] = pandas.read_csv(
        f'https://raw.githubusercontent.com/saspy-bffs/wuss-2024-python-
how/main/input_data/fakes/{name}-2024_08_04T13_09_15.csv',
        na_filter=False,
    )
```

1. In the code above, we're looping over the values in the list `['patients', 'dispenses', 'diagnoses']`, with the index variable `name` taking on each successive value in the list for each loop iteration. In addition, we dynamically insert the value of `name` into the string for each file URL using f-string notation.

In [4]:
```
# Create an empty dictionary object
df_dict = {}
# Loop over the input datasets, import them, and store each DataFrame in the dictionary
for name in ['patients', 'dispenses', 'diagnoses']:
    df_dict[name] = pandas.read_csv(
        f'https://raw.githubusercontent.com/saspy-bffs/wuss-2024-python-how/main/input_data/fakes/{name}-
        na_filter=False,
    )
```

Section 2. Data Exploration

Now that we've imported our three datasets, let's spend a few minutes inspecting metadata and generating summary tables and plots. Along the way, we may discover anomalies that should be fixed or excluded.

Example 2.1 Explore Patient Data

Type (or copy/paste) the following into the code cell immediately below, and then run that cell using Shift-Enter:

```
print('Confirm the patient dataset loaded correctly by printing the first few rows:')
display(patients_df.head(5))

print('\nExamine patient dataset column information:')
patients_df.info()

print(
    f"\nConfirm study_id can be used as a unique id in patients_df:",
    f"{confirm_unique_id_column(patients_df, 'study_id')}"
)
```

In [5]:
```
print('Confirm the patient dataset loaded correctly by printing the first few rows:')
display(patients_df.head(5))

print('\nExamine patient dataset column information:')
patients_df.info()

print(
    f"\nConfirm study_id can be used as a unique id in patients_df:",
    f"{confirm_unique_id_column(patients_df, 'study_id')}"
)
```

```
Confirm the patient dataset loaded correctly by printing the first few rows:
```

	study_id	first_name	middle_name	last_name	suffix	height	member	birth_date
0	S-0000	Albert	Craig	King	PhD	183.0	0	1940-05-20
1	S-0001	David	Ruben	White	MD	172.6	1	1941-04-13
2	S-0002	Daniel	Mary	Jones	DDS	172.7	1	1945-03-02
3	S-0003	Brian	Brian	Ramirez	Jr.	179.8	1	1940-04-25
4	S-0004	Damon	Roberto	Carpenter	PhD	179.7	1	1942-11-20

```
Examine patient dataset column information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4096 entries, 0 to 4095
Data columns (total 8 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   study_id     4096 non-null   object 
 1   first_name   4096 non-null   object 
 2   middle_name  4096 non-null   object 
 3   last_name    4096 non-null   object 
 4   suffix       4096 non-null   object 
 5   height       4096 non-null   float64
 6   member       4096 non-null   object 
 7   birth_date   4096 non-null   object 
dtypes: float64(1), object(7)
memory usage: 256.1+ KB

Confirm study_id can be used as a unique id in patients_df: True
```

Notes about Example 2.1.

1. The first 5 rows of `patients_df` and the column type information are displayed using the pandas `head` and `info` methods, respectively.

2. Note that the output from `head` displays row indices 0 through 4 since Python uses zero-based indexing.

3. Then, the `confirm_unique_id_column` function (defined in Section 0 above) is used to confirm there are no

missing or repeated values of `study_id`. By passing this data-integrity check, we can confidently use study IDs as unique identifiers for patients.

4. The pandas `head` method is the equivalent of PROC PRINT in SAS. For example, we could get similar output from the patients dataset as follows:

```
PROC PRINT DATA=patients_ds(obs=5);
RUN;
```

1. The pandas `info` method is the equivalent of PROC CONTENTS in SAS. For example, we could get similar metadata for the patients dataset as follows, with the `ORDER=varnum` used to print column information in column order (rather than the default alphabetical order):

```
PROC CONTENTS ORDER=varnum DATA=patients_ds;
RUN;
```

1. The `print` function is used throughout this notebook to create titles introducing other output. This usage is similar to the `title` statement in SAS, but `print` can also be used to display the values of objects like a DataFrame, similar to PROC PRINT or a `put` statement.

2. The `confirm_unique_id_column` allows us to check for duplicates without sorting, whereas in SAS we would typically use something like this:

```
PROC SORT DATA=patients_ds OUT=_NULL_ NODUPKEY;
    BY study_id;
RUN;
```

1. While SAS datasets are typically accessed from disk and processed row-by-row, DataFrames are loaded into memory all at once. This means values in DataFrames can be randomly accessed, but it also means the size of DataFrames can't grow beyond available memory.

2. As a reminder, the `display` method from `IPython` is equivalent to something like PROC PRINT or a `select` clause in PROC SQL in SAS.

Example 2.2 Check for Patient Data Anomalies

Type (or copy/paste) the following into the code cell immediately below, and then run that cell using Shift-Enter:

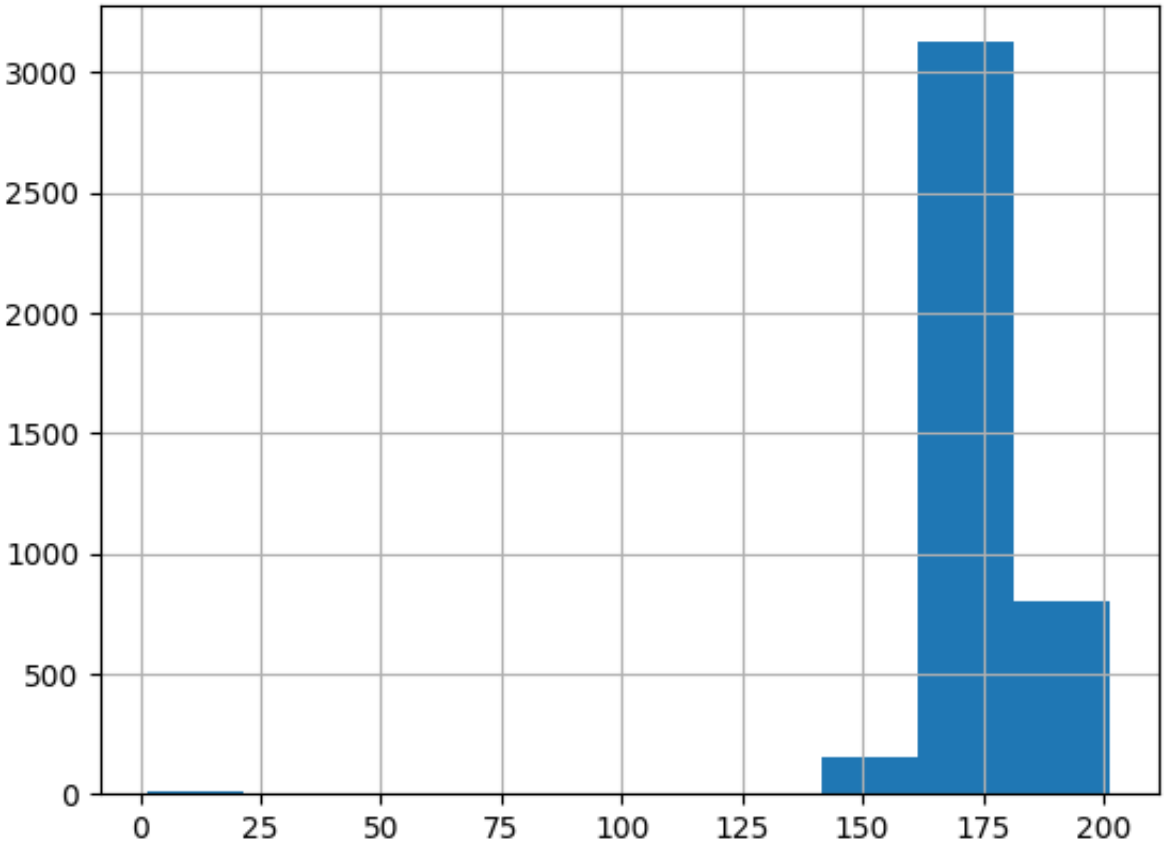
```
# Convert values in the birth_date column from strings to dates
patients_df['birth_date'] = pandas.to_datetime(patients_df['birth_date'])

print('\nExamine patient dataset column information to confirm conversion:')
patients_df.info()

print('Create two-way table: Patient year from birth_date column by member:')
display(pandas.crosstab(patients_df['birth_date'].dt.year, patients_df['member']))

print('\nExamine the distribution of patient heights as a histogram:')
height_hist = patients_df['height'].hist()
```

In [6]:
```
# Convert values in the birth_date column from strings to dates
patients_df['birth_date'] = pandas.to_datetime(patients_df['birth_date'])

print('\nExamine patient dataset column information to confirm conversion:')
patients_df.info()

print('Create two-way table: Patient year from birth_date column by member:')
display(pandas.crosstab(patients_df['birth_date'].dt.year, patients_df['member']))

print('\nExamine the distribution of patient heights as a histogram:')
height_hist = patients_df['height'].hist()
```

```
Examine patient dataset column information to confirm conversion:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4096 entries, 0 to 4095
Data columns (total 8 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   study_id     4096 non-null   object
 1   first_name   4096 non-null   object
 2   middle_name  4096 non-null   object
 3   last_name    4096 non-null   object
 4   suffix       4096 non-null   object
 5   height       4096 non-null   float64
 6   member       4096 non-null   object
 7   birth_date   4096 non-null   datetime64[ns]
dtypes: datetime64[ns](1), float64(1), object(6)
memory usage: 256.1+ KB
Create two-way table: Patient year from birth_date column by member:
```

member	0	1	N	Y
birth_date				
1840	5	5	0	0
1940	199	209	0	1
1941	210	195	1	1
1942	187	229	0	1
1943	208	228	3	1
1944	195	212	1	2
1945	206	194	0	2
1946	232	196	1	2
1947	209	180	0	2
1948	199	186	1	0
1949	203	189	1	0

Examine the distribution of patient heights as a histogram:



Notes about Example 2.2.

1. Looking at the output from `patients_df.info()` in Example 2.1, the `birth_date` column was imported with type `object`, which is a default catch-all when the values in a column aren't readily interpreted as numbers in `pandas`.

2. In order to work with the values of the `birth_date` column as calendar dates, we use the `to_datetime` method to convert them, and then we confirm the conversion process worked as expected with the `info` method.

3. Then, we build a two-way frequency table using the `pandas` method `crosstab`, counting the number of rows in the DataFrame by the `dt.year` component of `birth_date` and by `member` status.

4. A similar table could have been created in SAS using something like PROC FREQ or PROC TABULATE. Here's an example:

```
PROC FREQ DATA=patients_ds;
    TABLES birth_date*member;
    FORMAT birth_date year.;
RUN;
```

1. Note that two data anomalies are revealed in the output:
    - A small number of patients appear to have been born in the year 1840, which violates our assumption that all patients should have been born after January 1st, 1900, as defined in Section 0 above.
    - Some values of the `member` column are `'Y'` or `'N'` rather than the boolean values `0` and `1`.
2. Finally, we use a histogram to display patient `height` values, which are measured in centimeters. Looking closely at the plot, a small number of patients appear to have heights under 25 centimeters, which is quite unlikely!

3. An equivalent plot could have been generated in SAS using something like PROC SGPLOT:

```
PROC SGPLOT DATA=patients_ds;
    HISTOGRAM height;
RUN;
```

1. As a reminder, the `display` method from `IPython` is equivalent to something like PROC PRINT or a `select` clause in PROC SQL in SAS.

Example 2.3 Inspect Anomalous Patient Data

Type (or copy/paste) the following into the code cell immediately below, and then run that cell using Shift-Enter:

```
print('\nExamine records with anomalous values of patient birth_date values:')
display(patients_df[patients_df['birth_date'] < dob_lower_bound])

print('\nExamine records with anomalous values of patient height values:')
display(patients_df[patients_df['height'] < 25])
```

In [7]:
```
print('\nExamine records with anomalous values of patient birth_date values:')
display(patients_df[patients_df['birth_date'] < dob_lower_bound])

print('\nExamine records with anomalous values of patient height values:')
display(patients_df[patients_df['height'] < 25])
```

```
Examine records with anomalous values of patient birth_date values:
```

	study_id	first_name	middle_name	last_name	suffix	height	member	birth_date
309	S-0309	TEST		PATIENT		177.6	0	1840-12-31
455	S-0455	TEST		PATIENT		178.9	1	1840-12-31
946	S-0946	TEST		PATIENT		174.8	0	1840-12-31
1417	S-1417	TEST		PATIENT		182.3	1	1840-12-31
1525	S-1525	TEST		PATIENT		192.3	1	1840-12-31
1910	S-1910	TEST		PATIENT		176.6	1	1840-12-31
1926	S-1926	TEST		PATIENT		178.3	0	1840-12-31
2212	S-2212	TEST		PATIENT		182.6	0	1840-12-31
2231	S-2231	TEST		PATIENT		174.1	0	1840-12-31
2305	S-2305	TEST		PATIENT		176.1	1	1840-12-31

Examine records with anomalous values of patient height values:

	study_id	first_name	middle_name	last_name	suffix	height	member	birth_date
44	S-0044	Carol	Sarah	Jones	MD	1.648	0	1946-06-08
612	S-0612	Robert	David	Hill	MD	1.706	1	1947-06-04
911	S-0911	Lauren	Miguel	Williams	DDS	1.891	1	1946-12-05
1185	S-1185	Philip	James	Miller	DDS	1.724	0	1943-01-04
1674	S-1674	Vanessa	William	Taylor	DVM	1.803	0	1942-03-19
1724	S-1724	Sara	Andrew	Rivera	PhD	1.912	0	1946-09-17
1895	S-1895	Madison	James	Wood	DDS	1.798	1	1947-01-20
2154	S-2154	William	Anthony	Smith	DDS	1.809	1	1948-01-15
2497	S-2497	Abigail	Timothy	Clark	MD	1.674	1	1943-01-21
2740	S-2740	Lori	Peter	Smith	DDS	1.717	1	1949-03-01
3215	S-3215	Dwayne	Heidi	Jackson	MD	1.669	0	1948-08-12
3436	S-3436	Lauren	Steven	Potter	MD	1.742	1	1942-10-13
3775	S-3775	Patricia	Trevor	Mann	DVM	1.746	1	1946-05-05
3859	S-3859	Ricky	Melanie	Garrett	DDS	1.652	1	1944-11-23

Notes about Example 2.3.

1. In Example 2.2, we noted a small number of patients appear to have been born in the year 1840, which violates our assumption that all patients should have been born after January 1st, 1900, as defined in Section 0 above.

2. By filtering for patients were `birth_date` is less than `dob_lower_bound`, we see records that appear to have been generated as a test of a medical record system.

3. Similarly, we filter for patients where `height` is less than `25`, based on the behavior we saw in the histogram in Example 2.2. The resulting values for `height` appear to be in meters, rather than centimeters.

4. When we subset our DataFrames, note that we're putting an expression inside brackets, such as `[patients_df['birth_date'] < dob_lower_bound]`. Behind the scenes, this creates a "boolean mask" that tells `pandas` which rows to keep. The equivalent in SAS would be something like a `where` statement inside PROC PRINT:

```
PROC PRINT DATA=patients_ds(obs=5);
    WHERE birth_date < "&dob_lower_bound"d;
RUN;
```

1. As a reminder, the `display` method from `IPython` is equivalent to something like PROC PRINT or a `select` clause in PROC SQL in SAS.

Example 2.4 Explore Dispense Data

Type (or copy/paste) the following into the code cell immediately below, and then run that cell using Shift-Enter:

```
print('Confirm the dispenses dataset loaded correctly by printing the first few rows:')
display(dispenses_df.head(5))

print('\nExamine dispenses dataset column information:')
dispenses_df.info()

print(
    f"\nConfirm study_id can be used as a foreign key from dispenses_df to patients_df:",
    f"{confirm_foreign_key_column(dispenses_df, patients_df, 'study_id')}"
)
```

In [8]:
```
print('Confirm the dispenses dataset loaded correctly by printing the first few rows:')
display(dispenses_df.head(5))

print('\nExamine dispenses dataset column information:')
dispenses_df.info()

print(
    f"\nConfirm study_id can be used as a foreign key from dispenses_df to patients_df:",
    f"{confirm_foreign_key_column(dispenses_df, patients_df, 'study_id')}"
)
```

```
Confirm the dispenses dataset loaded correctly by printing the first few rows:
```

	study_id	dispense_date	drug_id	days_supply
0	S-0000	1995-06-03	Drug-A	90
1	S-0000	1995-08-31	Drug-A	60
2	S-0001	2007-12-30	Drug A	90
3	S-0001	2008-04-07	Drug A	60
4	S-0001	2008-06-01	Drug A	60

```
Examine dispenses dataset column information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 13072 entries, 0 to 13071
Data columns (total 4 columns):
 #   Column         Non-Null Count  Dtype 
---  ------         --------------  ----- 
 0   study_id       13072 non-null  object
 1   dispense_date  13072 non-null  object
 2   drug_id        13072 non-null  object
 3   days_supply    13072 non-null  int64 
dtypes: int64(1), object(3)
memory usage: 408.6+ KB

Confirm study_id can be used as a foreign key from dispenses_df to patients_df: True
```

Notes about Example 2.4.

1. Following the same pattern as in Example 2.1, the first 5 rows of `dispenses_df` and the column type information are displayed using the pandas `head` and `info` methods, respectively.

2. Then, the `confirm_foreign_key_column` function (defined in Section 0 above) is used to confirm there are no missing values of `study_id` and that all values of `study_id` appear in the corresponding column in `patients_df`. By passing this data-integrity check, we can confidently use study IDs to link dispenses to patients.

3. There are a variety of ways to perform a similar check in SAS. Here's an example in PROC SQL:

```
PROC SQL;
    SELECT study_id
    FROM dispenses
    WHERE study_id NOT IN (SELECT study_id FROM patients);
QUIT;
```

1. As a reminder, the `display` method from `IPython` is equivalent to something like PROC PRINT or a `select` clause in PROC SQL in SAS.

Example 2.5 Check for Dispense Data Anomalies

Type (or copy/paste) the following into the code cell immediately below, and then run that cell using Shift-Enter:

```
print('\nConvert values in the dispense_date column from strings to dates:')
dispenses_df['dispense_date'] = pandas.to_datetime(dispenses_df['dispense_date'])

print('\nExamine dispenses dataset column information to confirm conversion:')
dispenses_df.info()

print('Create three-way table: dispense year from dispense_date column by drug_id:')
display(
    pandas.crosstab(
        dispenses_df['dispense_date'].dt.year,
        columns=[dispenses_df['drug_id'],
        dispenses_df['days_supply']]
    )
)

print('\nExamine some example rows of dispenses with hyphenated drug_id values:')
display(dispenses_df[dispenses_df['drug_id'].str.contains('-')].head(5))
```

In [9]:
```
print('\nConvert values in the dispense_date column from strings to dates:')
dispenses_df['dispense_date'] = pandas.to_datetime(dispenses_df['dispense_date'])

print('\nExamine dispenses dataset column information to confirm conversion:')
dispenses_df.info()

print('Create three-way table: dispense year from dispense_date column by drug_id:')
display(
    pandas.crosstab(
        dispenses_df['dispense_date'].dt.year,
        columns=[dispenses_df['drug_id'],
        dispenses_df['days_supply']]
    )
)

print('\nExamine some example rows of dispenses with hyphenated drug_id values:')
display(dispenses_df[dispenses_df['drug_id'].str.contains('-')].head(5))
```

```
Convert values in the dispense_date column from strings to dates:

Examine dispenses dataset column information to confirm conversion:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 13072 entries, 0 to 13071
Data columns (total 4 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   study_id       13072 non-null  object
 1   dispense_date  13072 non-null  datetime64[ns]
 2   drug_id        13072 non-null  object
 3   days_supply    13072 non-null  int64
dtypes: datetime64[ns](1), int64(1), object(2)
memory usage: 408.6+ KB
Create three-way table: dispense year from dispense_date column by drug_id:
```

drug_id	Drug A			Drug B			Drug-A			Drug-B		
days_supply	30	60	90	30	60	90	30	60	90	30	60	90
dispense_date												

1990	85	85	88	76	91	77	8	5	4	3	7	6
1991	107	97	83	105	90	109	7	7	7	14	11	18
1992	99	94	89	101	111	108	15	12	20	9	9	9
1993	85	101	75	119	98	104	18	16	12	12	11	10
1994	110	92	98	83	91	88	11	8	12	7	3	4
1995	87	100	107	91	78	114	11	10	15	4	8	8
1996	97	111	100	117	136	90	2	9	11	10	6	11
1997	91	80	81	95	113	102	11	9	12	9	4	10
1998	84	90	96	103	96	94	9	3	10	5	10	9
1999	70	97	84	87	86	89	12	12	10	7	5	4
2000	104	107	107	103	119	117	17	14	12	5	7	14
2001	94	87	101	89	90	96	17	9	11	17	8	13
2002	91	91	103	100	66	73	13	7	10	7	12	13
2003	85	113	126	93	93	87	9	10	10	18	13	12
2004	92	103	96	128	111	112	11	8	16	7	13	8
2005	98	91	106	108	94	100	13	10	10	13	10	8
2006	99	85	97	89	103	91	12	9	12	9	6	13
2007	97	98	94	123	110	145	11	15	13	10	10	7
2008	106	113	106	112	118	116	11	10	10	9	12	16
2009	104	89	106	129	123	106	13	8	12	16	15	19

Examine some example rows of dispenses with hyphenated drug_id values:

	study_id	dispense_date	drug_id	days_supply
0	S-0000	1995-06-03	Drug-A	90
1	S-0000	1995-08-31	Drug-A	60
7	S-0003	1996-03-04	Drug-B	90
8	S-0003	1996-06-01	Drug-B	90
9	S-0003	1996-08-30	Drug-B	90

Notes about Example 2.5.

1. Looking at the output from `dispenses_df.info()` in Example 2.4, the `dispense_date` column was imported with type `object`, which is a default catch-all when the values in a column aren't readily interpreted as numbers in `pandas`.

2. In order to work with the values of the `dispense_date` column as calendar dates, we use the `to_datetime` method to convert them, and then we confirm the conversion process worked as expected with the `info` method.

3. Then, we build a three-way frequency table using the `pandas` method `crosstab`, counting the number of rows in the DataFrame by the `dt.year` component of `dispense_date`, by `drug_id`, and by `days_supply`.

4. A similar table could be created using PROC TABULATE:

```
PROC TABULATE DATA=dispenses;
    CLASS dispense_date drug_id*days_supply;
    TABLE dispense_date, drug_id days_supply;
    FORMAT dispense_date year.;
RUN;
```

1. Then, because a small number of `drug_id` values are written with a hyphen instead of a space, we display the first five rows of `dispenses_df` using the `pandas` method `head`, after filtering for rows with hyphenated `drug_id` values using the `str.contains` method.

2. There are a variety of ways to accomplish the same behavior in SAS, including using the `index` function in a `where` statement:

```
PROC PRINT DATA=patients_ds(obs=5);
    WHERE index(drug_id, '-');
RUN;
```

1. As a reminder, the `display` method from `IPython` is equivalent to something like PROC PRINT or a `select` clause in PROC SQL in SAS.

Example 2.6 Explore Diagnosis (Dx) Data

Type (or copy/paste) the following into the code cell immediately below, and then run that cell using Shift-Enter:

```
print('Confirm the diagnosis (dx) dataset loaded correctly by printing the first few rows:')
display(diagnoses_df.head(5))

print('\nExamine diagnosis (dx) dataset column information:')
diagnoses_df.info()

print(
    f"nConfirm study_id can be used as a foreign key from dispenses_df to patients_df:",
    f"{confirm_foreign_key_column(diagnoses_df, patients_df, 'study_id')}"
)
```

In [10]:
```
print('Confirm the diagnosis (dx) dataset loaded correctly by printing the first few rows:')
display(diagnoses_df.head(5))

print('\nExamine diagnosis (dx) dataset column information:')
diagnoses_df.info()

print(
    f"nConfirm study_id can be used as a foreign key from dispenses_df to patients_df:",
    f"{confirm_foreign_key_column(diagnoses_df, patients_df, 'study_id')}"
)
```

```
Confirm the diagnosis (dx) dataset loaded correctly by printing the first few rows:
```

	study_id	dx_date	dx_code	dx_name
0	S-0004	2014-07-11	I50.41	Acute combined systolic (congestive) and diast...
1	S-0004	2016-12-10	I50.42	Chronic combined systolic (congestive) and dia...
2	S-0014	2012-05-19	I50.33	Acute on chronic diastolic (congestive) heart ...
3	S-0017	2014-03-19	I97.130	Postprocedural heart failure following cardiac...
4	S-0017	2018-08-30	I50.3	Diastolic (congestive) heart failure

```
Examine diagnosis (dx) dataset column information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1785 entries, 0 to 1784
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype 
---  ------    --------------  ----- 
 0   study_id  1785 non-null   object
 1   dx_date   1785 non-null   object
 2   dx_code   1785 non-null   object
 3   dx_name   1785 non-null   object
dtypes: object(4)
memory usage: 55.9+ KB
nConfirm study_id can be used as a foreign key from dispenses_df to patients_df: True
```

Notes about Example 2.6.

1. Following the same pattern as in Examples 2.1 and 2.4, the first 5 rows of `diagnoses_df` and the column type information are displayed using the pandas `head` and `info` methods, respectively.

2. Then, following the same pattern as in Example 2.5, the `confirm_foreign_key_column` function (defined in Section 0 above) is used to confirm there are no missing values of `study_id` and that all values of `study_id` appear in the corresponding column in `patients_df`. By passing this data-integrity check, we can confidently use study IDs to link diagnoses (dx) to patients.

3. As a reminder, the `display` method from `IPython` is equivalent to something like PROC PRINT or a `select` clause in PROC SQL in SAS.

Example 2.7 Check for Diagnosis (Dx) Data Anomalies

Type (or copy/paste) the following into the code cell immediately below, and then run that cell using Shift-Enter:

```
# Convert values in dx_date column from strings to dates
diagnoses_df['dx_date'] = pandas.to_datetime(diagnoses_df['dx_date'])

print('\nExamine patient dataset column information to confirm conversion:')
diagnoses_df.info()

print('Create two-way table: dx_code by diagnosis year from dx_date column:')
display(pandas.crosstab(diagnoses_df['dx_code'], diagnoses_df['dx_date'].dt.year))

print('\nCount the number of rows for each combination of dx_code and dx_name:')
display(diagnoses_df[['dx_code', 'dx_name']].value_counts())
```

In [11]:
```
# Convert values in dx_date column from strings to dates
diagnoses_df['dx_date'] = pandas.to_datetime(diagnoses_df['dx_date'])

print('\nExamine patient dataset column information to confirm conversion:')
diagnoses_df.info()

print('Create two-way table: dx_code by diagnosis year from dx_date column:')
display(pandas.crosstab(diagnoses_df['dx_code'], diagnoses_df['dx_date'].dt.year))

print('\nCount the number of rows for each combination of dx_code and dx_name:')
display(diagnoses_df[['dx_code', 'dx_name']].value_counts())
```

```
Examine patient dataset column information to confirm conversion:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1785 entries, 0 to 1784
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   study_id  1785 non-null   object
 1   dx_date   1785 non-null   datetime64[ns]
 2   dx_code   1785 non-null   object
 3   dx_name   1785 non-null   object
dtypes: datetime64[ns](1), object(3)
memory usage: 55.9+ KB
Create two-way table: dx_code by diagnosis year from dx_date column:
```

dx_date	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019
dx_code										
I09.81	2	7	4	3	14	5	10	7	12	29
I11.0	5	4	4	6	8	7	3	6	8	26
I13.0	4	1	3	5	5	7	9	4	6	18
I13.2	3	3	5	1	9	7	8	7	9	18
I50.0	6	1	4	4	3	6	7	7	5	21
I50.1	2	2	7	4	4	3	4	4	6	21

I50.2	2	3	6	5	4	4	2	8	13	15
I50.20	3	7	7	5	3	6	10	9	6	28
I50.21	7	6	2	3	8	8	7	8	5	15
I50.22	3	2	6	8	6	8	9	7	2	24
I50.23	3	5	2	5	3	11	4	8	5	25
I50.3	4	2	3	5	1	3	7	6	11	25
I50.30	7	6	5	0	2	5	2	5	13	23
I50.31	1	1	6	6	8	6	7	5	10	30
I50.32	3	2	5	7	2	8	4	5	5	22
I50.33	2	2	8	7	4	6	9	7	1	25
I50.4	1	2	4	3	2	3	8	5	9	27
I50.40	2	6	2	6	8	8	1	11	10	22
I50.41	2	2	1	3	9	1	7	9	9	18
I50.42	2	3	3	7	2	6	7	10	12	29
I50.43	3	2	4	7	4	4	11	3	7	23
I50.9	3	5	4	5	5	5	10	6	7	31
I97.13	8	4	1	8	8	6	8	8	4	27
I97.130	1	3	4	3	2	7	6	8	12	18
I97.131	4	7	2	4	6	5	7	5	11	24

Count the number of rows for each combination of dx_code and dx_name:

dx_code	dx_name	count
I09.81	**Heart failure rheumatic (chronic) (inactive) (with chorea)**	93

I50.20	**Unspecified systolic (congestive) heart failure**	84
I97.13	**Postprocedural heart failure**	82
I50.9	**Heart failure, unspecified**	81
I50.42	**Chronic combined systolic (congestive) and diastolic (congestive) heart failure**	81
I50.31	**Acute diastolic (congestive) heart failure**	80
I11.0	**Hypertensive heart disease with (congestive) heart failure**	77
I50.40	**Unspecified combined systolic (congestive) and diastolic (congestive) heart failure**	76
I50.22	**Chronic systolic (congestive) heart failure**	75
I97.131	**Postprocedural heart failure following other surgery**	75
I50.23	**Acute on chronic systolic (congestive) heart failure**	71
I50.33	**Acute on chronic diastolic (congestive) heart failure**	71
I13.2	**Hypertensive heart and chronic kidney disease with heart failure and with stage 5 chronic kidney disease, or end stage renal disease**	70
I50.21	**Acute systolic (congestive) heart failure**	69
I50.43	**Acute on chronic combined systolic (congestive) and diastolic (congestive) heart failure**	68
I50.30	**Unspecified diastolic (congestive) heart failure**	68
I50.3	**Diastolic (congestive) heart failure**	67
I50.4	**Combined systolic (congestive) and diastolic (congestive) heart failure**	64
I50.0	**Congestive heart failure**	64
I97.130	**Postprocedural heart failure following cardiac surgery**	64
I50.32	**Chronic diastolic (congestive) heart failure**	63
I50.2	**Systolic (congestive) heart failure**	62
I13.0	**Hypertensive heart and chronic kidney disease with heart failure and stage 1 through stage 4 chronic kidney disease, or unspecified chronic kidney disease**	62

I50.41	**Acute combined systolic (congestive) and diastolic (congestive) heart failure**	61
I50.1	**Left ventricular failure**	57

dtype: int64

Notes about Example 2.7.

1. Looking at the output from `diagnoses_df.info()` in Example 2.6, the `dx_date` column was imported with type `object`, which is a default catch-all when the values in a column aren't readily interpreted as numbers in `pandas`.
2. In order to work with the values of the `dx_date` column as calendar dates, we use the `to_datetime` method to convert them, and then we confirm the conversion process worked as expected with the `info` method.
3. As our next step, we build a two-way frequency table using the `pandas` method `crosstab`, counting the number of rows in the DataFrame by `dx_code` and by the `dt.year` component of `dx_date`.
4. Similarly, we use the `pandas` method `value_counts` to build a table showing the number of rows by `dx_code` and `dx_name`. Since we expect there to be a one-to-one correspondence between codes and names, this table is displayed more like a dataset listing.
5. Fortunately, for once, there are no obvious data anomalies in a dataset. Hooray!
6. To generate similar tables in SAS, we could use something like PROC FREQ:

```
PROC FREQ DATA=diagnoses_ds;
    TABLES dx_code*dx_date;
    TABLES dx_code*dx_name / LIST;
    FORMAT dx_date year.;
RUN;
```

1. As a reminder, the `display` method from `IPython` is equivalent to something like PROC PRINT or a `select` clause in PROC SQL in SAS.

Section 3. Data Cleaning

Now that we've gotten to know our datasets, we can fix the data-quality issues we discovered in Section 2.

Example 3.1 Clean Patient Data

Type (or copy/paste) the following into the code cell immediately below, and then run that cell using Shift-Enter:

```
# Subset patient dataset to remove test data (based on implausible DOB values),
# map member values to be consistently numeric, and correct height values
# presumed to be in meters (rather than centimeters)
member_mapping = {
    '0': 0,
    'N': 0,
    '1': 1,
    'Y': 1,
}
patients_df_clean = (
    patients_df
    .query('birth_date  > @dob_lower_bound')
    .replace({'member': member_mapping})
    .reset_index(drop=True)
)
patients_df_clean.loc[patients_df_clean['height'] < 25, ['height']] *= 100
```

In [12]:
```
# Subset patient dataset to remove test data (based on implausible DOB values),
# map member values to be consistently numeric, and correct height values
# presumed to be in meters (rather than centimeters)
member_mapping = {
    '0': 0,
    'N': 0,
    '1': 1,
    'Y': 1,
}
patients_df_clean = (
    patients_df
    .query('birth_date  > @dob_lower_bound')
    .replace({'member': member_mapping})
    .reset_index(drop=True)
)
patients_df_clean.loc[patients_df_clean['height'] < 25, ['height']] *= 100
```

Notes about Example 3.1.

1. A dictionary named `member_mapping` is created, which maps string values to their numerical boolean equivalent:
   - The string values `'0'` and `'N'` map to the integer value `0`.
   - The string values `'1'` and `'Y'` map to the integer value `1`.
2. Dictionaries are a fundamental Python data structure, which are related to SAS formats and DATA step hash objects.
3. More generally, dictionaries are called *associative arrays* or *maps* because they map keys (appearing before the colons) to values (appearing after the colons). In other words, the value associated with each key can be accessed using bracket notation:
   - `member_mapping['0']` returns `0`
   - `member_mapping['N']` returns `0`
   - `member_mapping['1']` returns `1`
   - `member_mapping['Y']` returns `1`
4. We could obtain a similar mapping in SAS using a format or informat, as in this example:

```
PROC FORMAT;
    INVALUE member_mapping
        '0', 'N' = 0
        '1', 'Y' = 1;
RUN;
```

1. Then, to correct the data anomalies we identified in Examples 2.1-3, a new DataFrame named `patients_df_clean` is created using "method chaining," which means executing a series of "chained" operations on `patients_df`:

    - The `query` method subsets the input DataFrame by selecting rows where the value of `birth_date` is greater than the constant `dob_lower_bound` defined in Section 0. (Note the `@` symbol to distinguish a reference to a global object from a references to a DataFrame column. As used here, this is similar to referencing a SAS macro variable inside a DATA step or PROC SQL query.)
    - The `replace` method is used to replace the string values in the `member` column with boolean values as defined by the `member_mapping` dictionary created above.
    - The `reset_index` method resets the row index of the output DataFrame to be sequential starting at zero. When subsetting rows, it's common to reset the row index since `pandas` will otherwise retain the original indices.
2. Finally, the `.loc` method is used to operate on the subset of rows in `patients_df_clean` where heights are given in meters rather than centimeters. The `*=` operator is used to modify this column in place and convert from meters to centimeters by multiplying by 100.

3. Similar results could be obtained in SAS using a DATA step like the following, which references the `member_mapping` informat defined above:

```
DATA patients_ds_clean(RENAME=(member_numeric=member));
    SET patients_ds;
    WHERE birth_date > '01JAN1900'd;
    member_numeric = INPUT(member, member_mapping.);
    IF height < 25 THEN
        height = height*100;
    KEEP study_id first_name middle_name last_name suffix height member_numeric birth_date;
RUN;
```

Example 3.2 Inspect Cleaned Patient Data

Type (or copy/paste) the following into the code cell immediately below, and then run that cell using Shift-Enter:

```
print('\nConfirm no records remain with anomalous values of patient birth_date values:')
display(patients_df_clean[patients_df_clean['birth_date'] < dob_lower_bound])

print('\nConfirm member values have been properly converted:')
display(patients_df_clean['member'].value_counts())

print('\nExamine cleaned patient dataset column information:')
patients_df_clean.info()

print('\nConfirm no records with anomalous values of patient height remain:')
clean_height_hist = patients_df_clean['height'].hist()
```

In [13]:
```
print('\nConfirm no records remain with anomalous values of patient birth_date values:')
display(patients_df_clean[patients_df_clean['birth_date'] < dob_lower_bound])

print('\nConfirm member values have been properly converted:')
display(patients_df_clean['member'].value_counts())

print('\nExamine cleaned patient dataset column information:')
patients_df_clean.info()

print('\nConfirm no records with anomalous values of patient height remain:')
clean_height_hist = patients_df_clean['height'].hist()
```

```
Confirm no records remain with anomalous values of patient birth_date values:
```

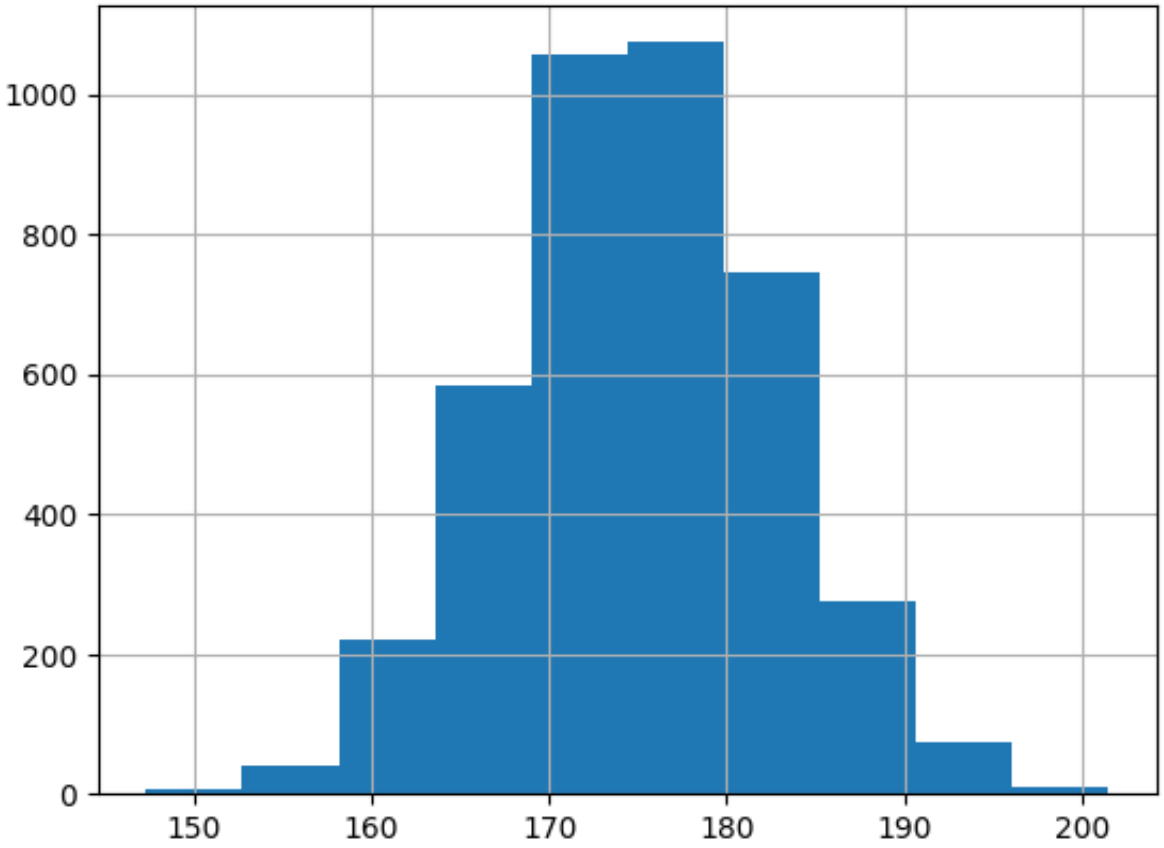
study_id	first_name	middle_name	last_name	suffix	height	member	birth_date

```
Confirm member values have been properly converted:
```

member	count
0	2056
1	2030

dtype: int64

```
Examine cleaned patient dataset column information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4086 entries, 0 to 4085
Data columns (total 8 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   study_id     4086 non-null   object
 1   first_name   4086 non-null   object
 2   middle_name  4086 non-null   object
 3   last_name    4086 non-null   object
 4   suffix       4086 non-null   object
 5   height       4086 non-null   float64
 6   member       4086 non-null   int64
 7   birth_date   4086 non-null   datetime64[ns]
dtypes: datetime64[ns](1), float64(1), int64(1), object(5)
memory usage: 255.5+ KB

Confirm no records with anomalous values of patient height remain:
```


1000
800
600
400
200
0
150
160
170
180
190
200

Notes about Example 3.2.

1. First, we display rows of `patients_df_clean` with `birth_date` values earlier than the lower bound defined in Section 0 above. Because we filtered out these rows in Example 3.1, the output should be an empty DataFrame.
2. Then, we display a frequency table for the `member` column to confirm that we properly mapped the original string values to the boolean values `0` and `1` in Example 3.1.
3. In addition, we display the column information for `patient_df_clean` to confirm the `member` column has been converted to an integer type.
4. Lastly, we display a histogram for `height` values, confirming values have been properly standardized to centimeter units.
5. As a reminder, output similar to `value_counts` could be obtained in SAS with PROC FREQ, output similar to `info` could be obtained with PROC CONTENTS, and output similar to `hist` could be obtained with PROC SGPLOT.
6. Also, the `display` method from `IPython` is equivalent to something like PROC PRINT or a `select` clause in PROC SQL in SAS.

Example 3.3 Clean Dispense Data

Type (or copy/paste) the following into the code cell immediately below, and then run that cell using Shift-Enter:

```
# Clean dispenses dataset to use consistent names for 'Drug A' and 'Drug B'
dispenses_df_clean = dispenses_df.copy()
dispenses_df_clean['drug_id'] = dispenses_df_clean['drug_id'].str.replace('-', ' ')

print('Confirm drug_id values have been properly converted:')
display(dispenses_df_clean['drug_id'].value_counts())
```

In [14]:
```
# Clean dispenses dataset to use consistent names for 'Drug A' and 'Drug B'
dispenses_df_clean = dispenses_df.copy()
dispenses_df_clean['drug_id'] = dispenses_df_clean['drug_id'].str.replace('-', ' ')

print('Confirm drug_id values have been properly converted:')
display(dispenses_df_clean['drug_id'].value_counts())
```

```
Confirm drug_id values have been properly converted:
```

drug_id	count
Drug B	6669
Drug A	6403

dtype: int64

Notes about Example 3.3.

1. In order to preserve our original dataset, we use the `copy` method to make a new copy of `dispenses_df` called `dispenses_df_clean`. (Note: Because complex objects like DataFrames can have multiple names in Python, it's not enough to use a statement like `dispenses_df_clean = dispenses_df`, which would result in the underlying DataFrame being referred to by both `dispenses_df_clean` and `dispenses_df`)

2. Then, to correct the data anomalies identified in Examples 2.4-5, we use the `str.replace` method to standardize the values in the `drug_id` column by replacing hyphens with spaces. After this step, all of the values in the column should be either `'Drug A'` and `'Drug B'`.

3. Similar results could be achieved in SAS using the `tranwrd` function inside of a DATA step:

```
DATA dispenses_ds_clean;
    SET dispenses_ds;
    drug_id = tranwrd(drug_id, '-', ' ');
RUN;
```

1. Finally, we use the `value_counts` method to create a frequency table for the values in the `drug_id` column to check the results of the transformation.

2. As a reminder, output similar to `value_counts` could be obtained in SAS with PROC FREQ.

3. Also, the `display` method from `IPython` is equivalent to something like PROC PRINT or a `select` clause in PROC SQL in SAS.

Section 4. Data Munging

With all of our data anomalies cleaned up or excluded, we're finally ready to create our analytic dataset!

This will include creating several new DataFrames and then combining everything together into a single DataFrame.

Example 4.1 Find Each Patient's Index Date

Type (or copy/paste) the following into the code cell immediately below, and then run that cell using Shift-Enter:

```
# Sort dispenses dataset by dispense date, subset to keep only the oldest
# dispense dates, and call this oldest dispense date the index data
first_dispenses_df = (
    dispenses_df_clean[['study_id', 'dispense_date']]
    .sort_values(by=['study_id', 'dispense_date'])
    .drop_duplicates(subset='study_id', keep='first')
    .rename(columns={'dispense_date': 'index_date'})
    .reset_index(drop=True)
)

print('Examine first_dispense_df:')
display(first_dispenses_df.head(5))

print('\nExamine index dates dataset column information:')
first_dispenses_df.info()
```

In [15]:
```
# Sort dispenses dataset by dispense date, subset to keep only the oldest
# dispense dates, and call this oldest dispense date the index data
first_dispenses_df = (
    dispenses_df_clean[['study_id', 'dispense_date']]
    .sort_values(by=['study_id', 'dispense_date'])
    .drop_duplicates(subset='study_id', keep='first')
    .rename(columns={'dispense_date': 'index_date'})
    .reset_index(drop=True)
)

print('Examine first_dispense_df:')
display(first_dispenses_df.head(5))

print('\nExamine index dates dataset column information:')
first_dispenses_df.info()
```

```
Examine first_dispense_df:
```

	study_id	index_date
0	S-0000	1995-06-03
1	S-0001	2007-12-30
2	S-0002	2007-08-25
3	S-0003	1996-03-04
4	S-0004	2005-07-03

```
Examine index dates dataset column information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4096 entries, 0 to 4095
Data columns (total 2 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   study_id    4096 non-null   object
 1   index_date  4096 non-null   datetime64[ns]
dtypes: datetime64[ns](1), object(1)
memory usage: 64.1+ KB
```

Notes about Example 4.1.

1. The *index date* is the date a patient becomes eligible to enter the study cohort.

2. In this study, the index date should be the first time a patient was dispensed Drug A or Drug B.

3. To create the `first_dispense_df` DataFrame, we use method chaining similar to Example 3.1:

    - First, we subset `dispensed_df_clean` to the columns `study_id` and `dispense_date` using the standard bracket notation `[]` for subsetting to specific columns in a DataFrame.
    - Note that the repeated brackets `[[ ]]` is not a typo! To subset a DataFrame to more than one column, we need to provide a list of column name, here `['study_id', 'dispense_date']`.
    - We then sort the DataFrame by the columns in the list `['study_id', 'dispense_date']`.

- Next, we drop rows with duplicate values of `study_id`, keeping only the first row for each Study ID.
- After that, we use a dictionary to say that the column name `dispense_date` should be changed to `index_date`.
- Finally, we reset the row index to be sequential starting at zero. (As a reminder, when subsetting rows, it's common to reset the row index since `pandas` will otherwise retain the original indices.)

4. These operations are equivalent to the following SAS code using a PROC SORT step and `first.` in a DATA step:

```
PROC SORT DATA=dispenses_ds_clean OUT=dispenses_ds_sorted;
    BY study_id dispense_date;
RUN;

DATA first_dispense_ds;
    SET dispenses_ds_sorted;
    BY study_id;
    IF first.study_id;
    KEEP study_id dispense_date;
    RENAME dispense_date=index_date;
RUN;
```

1. After all of that data manipulation, we display the first five rows of `first_dispense_df` and column info using the pandas methods `head` and `info`, respectively.

2. For extra credit, try executing the code below to use the `confirm_unique_id_column` function to check for duplicates, as in example 2.1:

```
print(
    f"study_id can be used as a unique id in first_dispenses_df:",
    f"{confirm_unique_id_column(first_dispenses_df, 'study_id')}"
)
```

In [16]:
```
print(
    f"study_id can be used as a unique id in first_dispenses_df:",
    f"{confirm_unique_id_column(first_dispenses_df, 'study_id')}"
)
```

```
study_id can be used as a unique id in first_dispenses_df: True
```

Example 4.2 Calculate Each Patient's Age at their Index Date

Type (or copy/paste) the following into the code cell immediately below, and then run that cell using Shift-Enter:

```
# Calculate patient age at index date
patients_age_df = pandas.merge(
    left=patients_df_clean,
    right=first_dispenses_df,
    how='inner',
    on=['study_id'],
    sort=True,
)
patients_age_df['age_at_index'] = calculate_age(patients_age_df, 'index_date', 'birth_date')

# Subset to columns of interest
columns_to_keep_list = ['study_id', 'birth_date', 'index_date', 'age_at_index', 'member',
'height']
patients_age_df = patients_age_df[columns_to_keep_list]

print('Confirm ages were calculated correctly by printing the first few rows:')
display(patients_age_df.head(5))

print('\nExamine patient ages dataset column information:')
patients_age_df.info()
```

In [17]:
```
# Calculate patient age at index date
patients_age_df = pandas.merge(
    left=patients_df_clean,
    right=first_dispenses_df,
    how='inner',
    on=['study_id'],
    sort=True,
)
patients_age_df['age_at_index'] = calculate_age(patients_age_df, 'index_date', 'birth_date')

# Subset to columns of interest
columns_to_keep_list = ['study_id', 'birth_date', 'index_date', 'age_at_index', 'member', 'height']
patients_age_df = patients_age_df[columns_to_keep_list]

print('Confirm ages were calculated correctly by printing the first few rows:')
display(patients_age_df.head(5))

print('\nExamine patient ages dataset column information:')
patients_age_df.info()
```

```
Confirm ages were calculated correctly by printing the first few rows:
```

	study_id	birth_date	index_date	age_at_index	member	height
0	S-0000	1940-05-20	1995-06-03	55	0	183.0
1	S-0001	1941-04-13	2007-12-30	66	1	172.6
2	S-0002	1945-03-02	2007-08-25	62	1	172.7
3	S-0003	1940-04-25	1996-03-04	55	1	179.8
4	S-0004	1942-11-20	2005-07-03	62	1	179.7

```
Examine patient ages dataset column information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4086 entries, 0 to 4085
Data columns (total 6 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   study_id      4086 non-null   object
 1   birth_date    4086 non-null   datetime64[ns]
 2   index_date    4086 non-null   datetime64[ns]
 3   age_at_index  4086 non-null   int32
 4   member        4086 non-null   int64
 5   height        4086 non-null   float64
dtypes: datetime64[ns](2), float64(1), int32(1), int64(1), object(1)
memory usage: 175.7+ KB
```

Notes about Example 4.2.

1. Now that we've determined the index date for each patient (i.e., the date they were first dispenses either Drug A or Drug B), the next step is to calculate each patient's age at their index date.

2. To start this process, we use the pandas `merge` method to combine the `patients_df_clean` DataFrame created in Example 3.1 and the `first_dispenses_df` DataFrame created in Example 4.1. In particular, we specify that the merge method should be an inner join by `study_id` and that the resulting table should be sorted by the column used for the join (i.e., `study_id`). In other words, we match up records from `patients_df_clean` and `first_dispenses_df` based on `study_id` and only keep patients having dipenses of Drug A or Drug B.

3. To get the same results in SAS, we could use a PROC SQL step like this, where we note there's more flexibility in specifying the column to sort the output table by:

```
PROC SQL;
  CREATE TABLE patients_age_ds AS
    SELECT A.*, B.*
    FROM patients_ds_clean AS A
    INNER JOIN first_dispenses_ds AS B
    ON study_id
```

```
    ORDER BY study_id;
QUIT;
```

1. Then, to calculate each patient's age at their index date, we use the function `calculate_age`, which was defined in Section 0 above. This function returns a `pandas` DataFrame column with the pairwise differences between the values in the `birth_date` and `index_date` columns, representing the age of the corresponding patient at the time they became eligible for the study by having a dispense of Drug A or Drug B.

2. Finally, we subset the columns in `patients_age_df` to the column names in the list `columns_to_keep_list`.

3. If we wanted to implement all three of these operations together in SAS, we could add additional components to the PROC SQL step above. Alternatively, we could use a PROC SORT step followed by DATA step:

```
PROC SORT data=patients_ds_clean OUT=patients_ds_sorted;
    BY study_id;
RUN;
DATA patients_age_ds;
    MERGE patients_ds_sorted(IN=in_cohort) first_dispense_ds(IN=has_dispense);
    BY study_id;
    IF NOT (in_cohort AND has_dispense)
        THEN DELETE;
    ELSE
        age_at_index = intck('year', birth_date, index_date, 'c');
    KEEP study_id index_date age_at_index member height;
RUN;
```

1. After all of this data manipulation, it's also helpful to display the first five rows of `patients_age_df` and column info using the pandas methods `head` and `info`, respectively.

2. For extra credit, try executing the code below to use the `confirm_unique_id_column` function to check for duplicates, as in example 2.1:

```
print(
    f"study_id can be used as a unique id in patients_age_df:",
```

```
    f"{confirm_unique_id_column(patients_age_df, 'study_id')}"
)
```

In [18]:
```
print(
    f"study_id can be used as a unique id in patients_age_df:",
    f"{confirm_unique_id_column(patients_age_df, 'study_id')}"
)
```

```
study_id can be used as a unique id in patients_age_df: True
```

Example 4.3 Calculate Each Patient's Total Supply Dispensed for each Drug

Type (or copy/paste) the following into the code cell immediately below, and then run that cell using Shift-Enter:

```
# Calculate total supply dispensed for each Drug
total_supply_df = (
    dispenses_df_clean
    .groupby(['study_id', 'drug_id'])['days_supply']
    .sum()
    .reset_index(drop=False)
    .rename(columns={'days_supply': 'total_supply'})
)

print('Confirm totals were calculated correctly by printing the first few rows:')
display(total_supply_df.head(5))

print('\nExamine total supply dispensed dataset column information:')
total_supply_df.info()
```

In [19]:
```
# Calculate total supply dispensed for each Drug
total_supply_df = (
    dispenses_df_clean
    .groupby(['study_id', 'drug_id'])['days_supply']
    .sum()
    .reset_index(drop=False)
    .rename(columns={'days_supply': 'total_supply'})
)

print('Confirm totals were calculated correctly by printing the first few rows:')
display(total_supply_df.head(5))

print('\nExamine total supply dispensed dataset column information:')
total_supply_df.info()
```

```
Confirm totals were calculated correctly by printing the first few rows:
```

	study_id	drug_id	total_supply
0	S-0000	Drug A	150
1	S-0001	Drug A	240
2	S-0002	Drug B	30
3	S-0003	Drug B	330
4	S-0004	Drug A	30

```
Examine total supply dispensed dataset column information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4096 entries, 0 to 4095
Data columns (total 3 columns):
 #   Column        Non-Null Count  Dtype 
---  ------        --------------  ----- 
 0   study_id      4096 non-null   object
 1   drug_id       4096 non-null   object
 2   total_supply  4096 non-null   int64 
dtypes: int64(1), object(2)
memory usage: 96.1+ KB
```

Notes about Example 4.3.

1. To create the `total_supply_df` DataFrame, we use method chaining similar to Examples 3.1 and 4.1:
   - First, we group the rows of `dispensed_df_clean` by columns `study_id` and `drug_id`, and then we subset the result to the column `days_supply` using the standard bracket notation `[]`. In other words, the `total_supply_df` DataFrame only has one column, and each row is indexed by a combination of values of `study_id` and `drug_id`.
   - We then sum the values of `days_supply` within each group so that the index of the resulting DataFrame will only have one row for each distinct combination of `study_id` and `drug_id`.
   - As our next step, we turn the multi-level row index values back into actual columns using the `reset_index` method with parameter `drop=False`.
   - And, finally, we use a dictionary to say that the column name `days_supply` should be changed to `total_supply`.
2. There are a variety of ways to sum within groups in SAS, including using a `group by` clause in PROC SQL or a PROC SUMMARY step like the following:

```
PROC SUMMARY DATA=dispenses_ds_clean NWAY;
    CLASS study_id drug_id;
    VAR days_supply;
    OUTPUT OUT=total_supply_ds(DROP=_:) SUM=total_supply;
RUN;
```

1. After all of that data manipulation, we display the first five rows of `first_dispense_df` and column info using the pandas methods `head` and `info`, respectively.

2. For extra credit, try executing the code below to use the `confirm_unique_id_column` function to check for duplicates, as in example 2.1:

```
print(
    f"study_id can be used as a unique id in total_supply_df:",
    f"{confirm_unique_id_column(total_supply_df, 'study_id')}"
```

```
)
```

In [20]:
```
print(
    f"study_id can be used as a unique id in total_supply_df:",
    f"{confirm_unique_id_column(total_supply_df, 'study_id')}"
)
```

```
study_id can be used as a unique id in total_supply_df: True
```

Example 4.4 Identify Patients with Heart Failure

Type (or copy/paste) the following into the code cell immediately below, and then run that cell using Shift-Enter:

```
# Get a list of patients with a heart failure (hf) diagnosis (dx)
hf_dx_outcome_df = (
    diagnoses_df
    .sort_values(by='study_id')
    .drop_duplicates(subset='study_id', keep='first')
    .filter(['study_id'])
    .reset_index(drop=True)
)

print('Confirm unique study IDs were selected correctly:')
display(hf_dx_outcome_df.head(5))

print('\nExamine outcomes dataset column information:')
hf_dx_outcome_df.info()
```

In [21]:
```
# Get a list of patients with a heart failure (hf) diagnosis (dx)
hf_dx_outcome_df = (
    diagnoses_df
    .sort_values(by='study_id')
    .drop_duplicates(subset='study_id', keep='first')
    .filter(['study_id'])
    .reset_index(drop=True)
)

print('Confirm unique study IDs were selected correctly:')
display(hf_dx_outcome_df.head(5))

print('\nExamine outcomes dataset column information:')
hf_dx_outcome_df.info()
```

```
Confirm unique study IDs were selected correctly:
```

	study_id
0	S-0004
1	S-0014
2	S-0017
3	S-0027
4	S-0028

```
Examine outcomes dataset column information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 874 entries, 0 to 873
Data columns (total 1 columns):
 #   Column    Non-Null Count  Dtype 
---  ------    --------------  ----- 
 0   study_id  874 non-null    object
dtypes: object(1)
memory usage: 7.0+ KB
```

Notes about Example 4.4.

1. To create the `hf_dx_outcome_df` DataFrame, we use method chaining similar to Examples 3.1, 4.1, and 4.3:
   - First, we sort `diagnoses_df` by `study_id`.
   - Then, we drop duplicates, keeping only the first record for each value of `study_id`.
   - Next, we subset the columns of the resulting DataFrame so that only `study_id` remains.
   - Finally, we reset the row index to be sequential starting at zero. (As a reminder, when subsetting rows, it's common to reset the row index since `pandas` will otherwise retain the original indices.)
2. There are a variety of ways to accomplish the same thing in SAS, including using a `select distinct` clause in PROC SQL or a PROC SORT step like the following:

```
PROC SORT DATA=diagnoses_ds OUT=hf_dx_outcome_ds(keep=study_id) NODUPKEY;
    BY study_id;
RUN;
```

1. After all of that data manipulation, we display the first five rows of `hf_dx_outcome_df` and column info using the pandas methods `head` and `info`, respectively.

2. For extra credit, try executing the code below to use the `confirm_unique_id_column` function to check for duplicates, as in example 2.1:

```
print(
    f"study_id can be used as a unique id in hf_dx_outcome_df:",
    f"{confirm_unique_id_column(hf_dx_outcome_df, 'study_id')}"
)
```

In [22]:
```
print(
    f"study_id can be used as a unique id in hf_dx_outcome_df:",
    f"{confirm_unique_id_column(hf_dx_outcome_df, 'study_id')}"
)
```

```
study_id can be used as a unique id in hf_dx_outcome_df: True
```

Example 4.5 Create Final Analytic Dataset

Type (or copy/paste) the following into the code cell immediately below, and then run that cell using Shift-Enter:

```
# Create final analytic dataset
analytic_df = pandas.merge(
    left=patients_age_df,
    right=total_supply_df,
    how='left',
    on=['study_id'],
    sort=True,
)
analytic_df = (
    analytic_df
    .assign(drug_a_supply=(analytic_df['drug_id'] == 'Drug A')*analytic_df['total_supply'])
    .assign(drug_b_supply=(analytic_df['drug_id'] == 'Drug B')*analytic_df['total_supply'])

.assign(dx_outcome=analytic_df['study_id'].isin(hf_dx_outcome_df['study_id']).astype(int))
    .filter(['study_id', 'dx_outcome', 'member', 'age_at_index', 'height', 'drug_a_supply',
'drug_b_supply'])
)

print('Confirm analytic dataset was formed correctly by printing the first few rows:')
display(analytic_df.head(5))

print('\nExamine analytic dataset column information:')
analytic_df.info()
```

In [23]:
```
# Create final analytic dataset
analytic_df = pandas.merge(
    left=patients_age_df,
    right=total_supply_df,
    how='left',
    on=['study_id'],
    sort=True,
)
analytic_df = (
    analytic_df
    .assign(drug_a_supply=(analytic_df['drug_id'] == 'Drug A')*analytic_df['total_supply'])
    .assign(drug_b_supply=(analytic_df['drug_id'] == 'Drug B')*analytic_df['total_supply'])
    .assign(dx_outcome=analytic_df['study_id'].isin(hf_dx_outcome_df['study_id']).astype(int))
    .filter(['study_id', 'dx_outcome', 'member', 'age_at_index', 'height', 'drug_a_supply', 'drug_b_suppl
)

print('Confirm analytic dataset was formed correctly by printing the first few rows:')
display(analytic_df.head(5))

print('\nExamine analytic dataset column information:')
analytic_df.info()
```

```
Confirm analytic dataset was formed correctly by printing the first few rows:
```

	study_id	dx_outcome	member	age_at_index	height	drug_a_supply	drug_b_supply
0	S-0000	0	0	55	183.0	150	0
1	S-0001	0	1	66	172.6	240	0
2	S-0002	0	1	62	172.7	0	30
3	S-0003	0	1	55	179.8	0	330
4	S-0004	1	1	62	179.7	30	0

```
Examine analytic dataset column information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4086 entries, 0 to 4085
Data columns (total 7 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   study_id        4086 non-null   object
 1   dx_outcome      4086 non-null   int64
 2   member          4086 non-null   int64
 3   age_at_index    4086 non-null   int32
 4   height          4086 non-null   float64
 5   drug_a_supply   4086 non-null   int64
 6   drug_b_supply   4086 non-null   int64
dtypes: float64(1), int32(1), int64(4), object(1)
memory usage: 207.6+ KB
```

Notes about Example 4.5.

1. Now that we've built all of the pieces needed, we're ready to build our final, analytic dataset!

2. To get started, we use the pandas `merge` method to combine the `patients_age_df` DataFrame created in Example 4.2 and the `total_supply_df` DataFrame created in Example 4.3. In particular, we specify that the merge method should be a left join by `study_id` and that the resulting table should be sorted by the column used for the join (i.e., `study_id`). In other words, we match up records from `patients_age_df` and `total_supply_df` based on `study_id`, keeping all rows in `patients_age_df`, even if there are no matching rows in `total_supply_df`.

3. Next, we refine the resulting `analytic_df` DataFrame using method chaining similar to Examples 3.1, 4.1, 4.3, and 4.4:

    - By repeatedly using the `assign` method, we create the three new columns `drug_a_supply`, `drug_b_supply`, and `dx_outcome`.
    - In the creation of `drug_a_supply`, we create a boolean mask (i.e., a single-column DataFrame whose only values are `True` and `False`) based on the calculation `analytic_df['drug_id'] == 'Drug A'`. Then, since `True` is equivalent to `1` and `False` is equivalent to `0`, we can form the pairwise product with the

column `analytic_df['total_supply']`, effectively pivoting our data so that `'Drug A'` becomes a column rather than a value in a row.
    - The creation of `drug_b_supply` is similar, but for `'Drug B'`.
    - The creation of `dx_outcome` is based on the boolean value of whether a value of `study_id` is in the DataFrame `hf_dx_outcome_df` created in Example 4.4, which is then explicitly type case as an integer value. This will allow us to use `dx_outcome` as our binary outcome variable in a logistic regression model.
    - Finally, we subset the resulting DataFrame to the only columns we'll need for our logistic regression model.
4. There are a variety of ways to accomplish the same thing in SAS, including using a DATA step like the following:

```
DATA analytic_ds;
    MERGE patients_age_ds(IN=pop) total_supply_ds hf_dx_outcome_ds(IN=has_dx);
    BY study_id;
    drug_a_supply = ifn(drug_id EQ 'Drug A', total_supply, 0);
    drug_b_supply = ifn(drug_id EQ 'Drug B', total_supply, 0);
    dx_outcome = has_dx;
    IF pop;
    KEEP study_id dx_outcome member age_at_index  height drug_a_supply drug_b_supply;
RUN;
```

1. After all of that data manipulation, we display the first five rows of `analytic_df` and column info using the pandas methods `head` and `info`, respectively.

2. For extra credit, try executing the code below to use the `confirm_unique_id_column` function to check for duplicates, as in example 2.1:

```
print(
    f"study_id can be used as a unique id in analytic_df:",
    f"{confirm_unique_id_column(analytic_df, 'study_id')}"
)
```

In [24]:
```
print(
    f"study_id can be used as a unique id in analytic_df:",
    f"{confirm_unique_id_column(analytic_df, 'study_id')}"
)
```

```
study_id can be used as a unique id in analytic_df: True
```

Section 5. Data Management

Given how much work goes into building analytic datasets, it's usually a good idea to save them to permanent files so that analyses can be more easily reproduced later.

Example 5.1 Save Analytic Dataset to Disk in Multiple Formats

Type (or copy/paste) the following into the code cell immediately below, and then run that cell using Shift-Enter:

```
# Create output file paths
csv_output_filepath = output_dir / 'analytic_df.csv'
parquet_output_filepath = output_dir / 'analytic_df.parquet'

# Write analytic dataset to disk as a .csv file
analytic_df.to_csv(csv_output_filepath, index=False)

# Write analytic dataset to disk as a .parquet file
analytic_df.to_parquet(parquet_output_filepath, engine='pyarrow')
```

In [25]:
```
# Create output file paths
csv_output_filepath = output_dir / 'analytic_df.csv'
parquet_output_filepath = output_dir / 'analytic_df.parquet'

# Write analytic dataset to disk as a .csv file
analytic_df.to_csv(csv_output_filepath, index=False)

# Write analytic dataset to disk as a .parquet file
analytic_df.to_parquet(parquet_output_filepath, engine='pyarrow')
```

Notes about Example 5.1.

1. First, we extend the `output_dir` path object created in Section 0 above to build two file paths: One for a .csv file and one for a .parquet file. Note, in particular, the use of a forward slash `/` to connect path components, which is borrowed from the way paths are delimited in Linux or Unix-like operating systems.

2. The `analytic_df` DataFrame is then written to disk using the `to_csv` method, with the `index=False` parameter used to suppress the inclusion of the index. As with `from_csv` in Section 1, `pandas` will only create output if an error occurs.

3. Similarly, the `analytic_df` DataFrame is written to disk using the `to_parquet` method, with the `engine='pyarrow'` parameter telling `pandas` to use the `pyarrow` package for the conversion process. (Note: The package `pyarrow` is not included in the Python standard library, but it is included by default in Colab, which is why we didn't need to install it.)

4. Unlike SAS, Python has no official standard file format for saving datasets to disk. It's common to see .csv files, to the point that CSV can feel like a de facto standard, but the Parquet format has the advantage of preserving column type information and potentially making read/write operations faster.

5. In SAS, on the other hand, it would be common to write a permanent SAS dataset to disk using a libref, along with any other desired file formats, using code like the following:

```
* save CSV file;
```

```
PROC EXPORT DATA=analytic_ds
    OUTFILE="analytic_ds.csv"
    DBMS=csv
    REPLACE;
RUN;

* save permanent SAS dataset;
DATA sasds.analytic_ds;
    SET analytic_ds;
RUN;
```

1. For extra credit, try executing the shell commands below to view the contents of the files written to disk (where, as in Section 0, we use the `!` command to submit the statements to the underlying operating system):

```
!echo 'Examine analytic_df.csv:'
!head /content/permanent_datasets/analytic_df.csv

!echo # this empty echo command creates a blank line between output

!echo 'Examine analytic_df.parquet:'
!parquet-tools show --head 10 /content/permanent_datasets/analytic_df.parquet
```

In [26]:
```
!echo 'Examine analytic_df.csv:'
!head /content/permanent_datasets/analytic_df.csv

!echo # this empty echo command creates a blank line between output

!echo 'Examine analytic_df.parquet:'
!parquet-tools show --head 10 /content/permanent_datasets/analytic_df.parquet
```

```
Examine analytic_df.csv:
study_id,dx_outcome,member,age_at_index,height,drug_a_supply,drug_b_supply
S-0000,0,0,55,183.0,150,0
S-0001,0,1,66,172.6,240,0
S-0002,0,1,62,172.7,0,30
S-0003,0,1,55,179.8,0,330
S-0004,1,1,62,179.7,30,0
S-0005,0,0,57,167.9,0,150
S-0006,0,0,52,168.4,60,0
S-0007,0,1,61,168.9,60,0
S-0008,0,1,51,184.8,180,0

Examine analytic_df.parquet:
```

study_id	dx_outcome	member	age_at_index	height	drug_a_supply	drug_b_supply
S-0000	0	0	55	183	150	0
S-0001	0	1	66	172.6	240	0
S-0002	0	1	62	172.7	0	30
S-0003	0	1	55	179.8	0	330
S-0004	1	1	62	179.7	30	0
S-0005	0	0	57	167.9	0	150
S-0006	0	0	52	168.4	60	0
S-0007	0	1	61	168.9	60	0
S-0008	0	1	51	184.8	180	0
S-0009	0	0	47	170.5	0	180

Section 6. Analysis

After all of that work, we're finally ready to ask the question that motivated this project in the first place: Is the risk of heart failure related to receipt of either Drug A or Drug B?

To attempt to answer this question, we will use logistic regression to model the probability of heart failure based on the total supply of each drug, while also controlling for the patient characteristics in the columns `member`, `age_at_index`, and `height`.

Example 6.1 Build a Logistic Regression Model

Type (or copy/paste) the following into the code cell immediately below, and then run that cell using Shift-Enter:

```
logit_model = logit(
    formula='dx_outcome ~ member + age_at_index + height + drug_a_supply + drug_b_supply',
    data=analytic_df,
).fit()

print('\nModel summary information:')
print(logit_model.summary2())
```

In [27]:
```
logit_model = logit(
    formula='dx_outcome ~ member + age_at_index + height + drug_a_supply + drug_b_supply',
    data=analytic_df,
).fit()

print('\nModel summary information:')
print(logit_model.summary2())
```

```
Optimization terminated successfully.
         Current function value: 0.517428
         Iterations 5

Model summary information:
                          Results: Logit
================================================================
Model:              Logit            Method:           MLE
Dependent Variable: dx_outcome       Pseudo R-squared: 0.002
Date:               2024-09-04 23:45 AIC:              4240.4204
No. Observations:   4086             BIC:              4278.3123
Df Model:           5                Log-Likelihood:   -2114.2
Df Residuals:       4080             LL-Null:          -2118.4
Converged:          1.0000           LLR p-value:      0.14112
No. Iterations:     5.0000           Scale:            1.0000
----------------------------------------------------------------
                 Coef.  Std.Err.    z    P>|z|   [0.025  0.975]
----------------------------------------------------------------
Intercept        -2.1223   0.9466 -2.2420 0.0250 -3.9777 -0.2670
member            0.0795   0.0765  1.0396 0.2985 -0.0704  0.2295
age_at_index     -0.0029   0.0058 -0.4969 0.6192 -0.0144  0.0085
height            0.0045   0.0051  0.8923 0.3722 -0.0054  0.0145
drug_a_supply     0.0005   0.0004  1.2761 0.2019 -0.0003  0.0013
drug_b_supply     0.0010   0.0004  2.4999 0.0124  0.0002  0.0017
================================================================
```

Notes about Example 6.1.

1. The `logit_model` object is created using the `fit` method of the `logit` object from the `statsmodels` package, which we imported in Section 0 above.

2. As parameters, we provide the `analytic_df` DataFrame as the input dataset, along with a string specifying the dependent variable (`dx_outcome`) and the explanatory variables (`member` , `age_at_index` , `height` , `drug_a_supply` , and `drug_b_supply`) using syntax similar to the `lm` function in R.

3. Then, using the `summary2` method, we print a multi-line string summarizing the logistic regression model.

4. Based on this output, there appears to be a weak, but still statistically significant, relationship between `drug_b_supply` and the heart failure outcome at the 95% confidence level since the associated P-value is less than 0.05.

5. However, none of the other covariates appear to be statistically significant, implying there may be an increased risk of heart failure associated with receipt of Drug B.

6. The equivalent analysis in SAS could be performed with a variety of analytical procedure, including the following example with PROC LOGISTIC:

```
PROC LOGISTIC DATA=analytic_ds;
    MODEL dx_outcome(EVENT='1') = member age_at_index height drug_a_supply drug_b_supply;
RUN;
```

1. As a side note, one of the most popular Python package for logistic regression is `scikit-learn` . However, `scikit-learn` is more geared towards predictive-analytics tasks like building machine-learning models, whereas `statsmodels` provides more conventional explanatory modeling tools.

2. For extra credit, try executing the code below to get odds ratios for the explanatory variables in our model:

```
from numpy import exp
print(exp(logit_model.summary2().tables[1]['Coef.'][1:]))
```

In [28]:
```
from numpy import exp
print(exp(logit_model.summary2().tables[1]['Coef.'][1:]))
```

```
member           1.082800
age_at_index     0.997102
height           1.004533
drug_a_supply    1.000501
drug_b_supply    1.000970
Name: Coef., dtype: float64
```

Section 7. Extra Credit

In a real academic study, it's typical to create a summary table combining descriptive statistics for the explanatory variables in a model with statistical output such as P-values or odds ratios.

For extra credit, you're encouraged to work through the steps below to a table that includes the P-values calculated in Section 6.1 above.

Example 7.1 Build a Fancy Output Table

Instructions: Click anywhere in the code cell below, and run the cell by pressing Shift-Enter or by clicking the triangular "play" icon in the upper left corner of the cell.

In [29]:
```
# Put together summary information for explanatory variables by building
# and then stitching together the contents from several summary tables

numeric_explanatory_variable_list = ['age_at_index', 'height', 'drug_a_supply', 'drug_b_supply']

number_of_pts_by_dx_outcome = (analytic_df['dx_outcome'].value_counts())

number_of_members_df = (
    pandas.crosstab(analytic_df['member'], analytic_df['dx_outcome'])
    .set_axis(['No', 'Yes'])
```

```
    .assign(overall=analytic_df['member'].value_counts().values)
    .rename_axis(None, axis=1)
)

for column_name in [0, 1, 'overall']:
    number_of_members_df[column_name] = [
        f'{member_count:,} ({member_count/sum(number_of_members_df[column_name]):.1%})'
        for member_count in number_of_members_df[column_name]
    ]

numerical_summary_df = (
    analytic_df
    .groupby('member')[numeric_explanatory_variable_list].agg(['mean', 'std'])
    .transpose()
    .assign(
        overall=(
            analytic_df[numeric_explanatory_variable_list]
            .agg(['mean', 'std'])
            .transpose()
            .stack()
        )
    )
)

summary_statistics_df = pandas.DataFrame(index=numeric_explanatory_variable_list)
for column_name in [0, 1, 'overall']:
    summary_statistics_df[column_name] = [
        (
            f'{numerical_summary_df.loc[(variable_name, "mean"),column_name]:.1f} '
            f'({numerical_summary_df.loc[(variable_name, "std"),column_name]:.1f})'
        )
        for variable_name in numeric_explanatory_variable_list
    ]

combined_statistics_df = (
    pandas.merge(
        pandas.concat([number_of_members_df, summary_statistics_df]),
        logit_model.summary2().tables[1]['P>|z|'],
        how='left',
```

```
        left_index=True,
        right_index=True,
    )
)
combined_statistics_df.loc['No', 'P>|z|'] = logit_model.summary2().tables[1].loc['member', 'P>|z|']
combined_statistics_df = (
    combined_statistics_df
    .rename(
        index={
            'No': '\tNo',
            'Yes': '\tYes',
            'age_at_index': 'Age at Index, mean (std)',
            'height': 'Height, mean (std)',
            'drug_a_supply': 'Drug A, mean (std)',
            'drug_b_supply': 'Drug B, mean (std)',
        }
    )
    .reset_index()
)

# Confirm the combined statistics summary tables was formed correctly
display(combined_statistics_df)

# Build a representation of the statistics summary tables with formatting
combined_statistics_gt = (
    GT(combined_statistics_df)
    .tab_spanner(label="Heart Failure Diagnosis", columns=['0', '1'])
    .cols_move_to_start(columns=["index", "1", "0"])
    .cols_label(
        **{
            'index': 'Membership, n (%)',
            '0': f'No Heart Failure, N={number_of_pts_by_dx_outcome.loc[0]:,}',
            '1': f'Any Heart Failure, N={number_of_pts_by_dx_outcome.loc[1]:,}',
            'overall': f'Overall, N={sum(number_of_pts_by_dx_outcome):,}',
            'P>|z|': 'P-value',
        }
    )
    .fmt_number(columns='P>|z|', decimals=3, use_seps=False)
    .tab_style(
```

```
        style=style.text(whitespace="pre"),
        locations=loc.body(columns="index"),
    )
)

# Print the contents of formatted statistics summary tables
combined_statistics_gt.show()
```

	index	0	1	overall	P>|z|
0	\tNo	1,630 (50.7%)	426 (48.9%)	2,056 (50.3%)	0.298532
1	\tYes	1,584 (49.3%)	446 (51.1%)	2,030 (49.7%)	NaN
2	Age at Index, mean (std)	54.5 (6.5)	54.7 (6.6)	54.6 (6.5)	0.619249
3	Height, mean (std)	175.0 (7.6)	175.1 (7.5)	175.1 (7.5)	0.372222
4	Drug A, mean (std)	91.1 (120.2)	97.7 (125.0)	94.4 (122.6)	0.201910
5	Drug B, mean (std)	101.6 (124.2)	93.1 (119.6)	97.4 (122.0)	0.012422

```
/usr/local/lib/python3.10/dist-packages/great_tables/_tbl_data.py:639: UserWarning: pandas DataFrame con
tains non-string column names. Coercing to strings. Here are the first few non-string columns:

  * Position 1: 0
  * Position 2: 1
  warnings.warn(
```

	Heart Failure Diagnosis			
Membership, n (%)	Any Heart Failure, N=872	No Heart Failure, N=3,214	Overall, N=4,086	P-value
No	426 (48.9%)	1,630 (50.7%)	2,056 (50.3%)	0.299
Yes	446 (51.1%)	1,584 (49.3%)	2,030 (49.7%)	
Age at Index, mean (std)	54.7 (6.6)	54.5 (6.5)	54.6 (6.5)	0.619
Height, mean (std)	175.1 (7.5)	175.0 (7.6)	175.1 (7.5)	0.372
Drug A, mean (std)	97.7 (125.0)	91.1 (120.2)	94.4 (122.6)	0.202
Drug B, mean (std)	93.1 (119.6)	101.6 (124.2)	97.4 (122.0)	0.012

Notes about Example 7.1.

1. A series of Pandas operations are used to gather frequencies for the categorical explanatory variable `member`, along with summary statistics for the numeric explanatory variables `age_at_index`, `height`, `drug_a_supply`, and `drug_b_supply`.
2. These frequencies and summary statistics are then merged into the `combined_statistics_df` DataFrame and combined with P-values from the output of the logistic regression model created in Example 6.1.
3. Then, methods from the `great_tables` package are used to create column headers and control the appearance of the final output report, which includes summary statistics for all of the explanatory variables in our logistic regression model, along with P-values for each variable.
4. Finally, the resulting table is displayed.
5. The equivalent workflow in SAS would typically involve calculating summary statistics with PROC FREQ and PROC MEANS steps, before combining them with statistical output using a DATA step or a PROC SQL step.

Wrapping Up: Call to Action!

Want some ideas for what to do next? Here are our suggestions:

1. Continue learning Python.
    - For general programming, we recommend starting with these:
        - Automate the Boring Stuff with Python, a free online book with beginner-friendly explanations of fundamental Python concepts, along with numerous hands-on projects
        - Fluent Python, which provided a deep dive into Intermediate to Advanced Python concepts
    - For data science, we recommend starting with these:
        - A Whirlwind Tour of Python, a free online book with coverage of essential Python features commonly used in data projects
        - Python for Data Analysis, which provided a deep dive into the `pandas` package by its creator, Wes McKinney
2. Keep in touch for follow-up questions/discussion (one of our favorite parts of teaching!) using isaiah.lankham@gmail.com and matthew.t.slaughter@gmail.com
3. If you have a GitHub account (or don't mind creating one), you can also chat with us on Gitter at https://gitter.im/saspy-bffs/community