3. Data Preparation

3.1. Data Preparation with Pandas

This chapter will show you how to use the pandas package to import and preprocess data. Preprocessing is the process of pre-analyzing data before converting it to a standard and normalized format. The following are some of the aspects of preprocessing:

- missing values
- data normalization
- data standardization
- data binning

We'll simply be dealing with missing values in this session.

3.1.1. Importing data

We will utilize the Iris dataset in this tutorial, which can be downloaded from the UCI Machine Learning Repository, https://archive.ics.uci.edu/ml/datasets/iris.

In the pattern recognition literature, this is probably the most well-known database. Fisher's paper is considered a classic in the subject and is still cited frequently. (See, for example, Duda & Hart.) The data collection has three classes, each with 50 instances, each referring to a different species of iris plant. The three classes include

- Iris Setosa
- Iris Versicolour
- Iris Virginica.

To begin, use the pandas library to import data and transform it to a dataframe.

```
import pandas as pd
```

```
#iris = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data', header=None,
#                names = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width',
'class'])

iris = pd.read_csv('/Users/Kaemyuijang/SCMA248/Data/iris.data', header=None, names =
['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'class'])


#url = 'https://raw.githubusercontent.com/pairote-sat/SCMA248/main/Data/iris.data'

#iris = pd.read_csv(url, header=None, names = ['sepal_length', 'sepal_width',
'petal_length', 'petal_width', 'class'])

#iris = pd.read_csv('https://raw.githubusercontent.com/pairote-
sat/SCMA248/main/Data/iris.data', header=None,
#                names = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width',
'class'])

type(iris)
```

```
pandas.core.frame.DataFrame
```

Here we specify whether there is a header (`header`) and the variable names (using `names` and a list).

The resulting object `iris` is a pandas DataFrame.

We will print the first 10 rows and the last 10 rows of the dataset using the head(10) method to get an idea of its contents.

Contents

```
iris.head(10)
```

	sepal_length	sepal_width	petal_length	petal_width	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
5	5.4	3.9	1.7	0.4	Iris-setosa
6	4.6	3.4	1.4	0.3	Iris-setosa
7	5.0	3.4	1.5	0.2	Iris-setosa
8	4.4	2.9	1.4	0.2	Iris-setosa
9	4.9	3.1	1.5	0.1	Iris-setosa

```
iris.tail(10)
```

	sepal_length	sepal_width	petal_length	petal_width	class
140	6.7	3.1	5.6	2.4	Iris-virginica
141	6.9	3.1	5.1	2.3	Iris-virginica
142	5.8	2.7	5.1	1.9	Iris-virginica
143	6.8	3.2	5.9	2.3	Iris-virginica
144	6.7	3.3	5.7	2.5	Iris-virginica
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

To get the names of the columns (the variable names), you can use `columns` method.

```
iris.columns
```

```
Index(['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'class'],
dtype='object')
```

To extract the class column, you can simply use the following commands:

```
iris['class']
```

```
0         Iris-setosa
1         Iris-setosa
2         Iris-setosa
3         Iris-setosa
4         Iris-setosa
            ...
145    Iris-virginica
146    Iris-virginica
147    Iris-virginica
148    Iris-virginica
149    Iris-virginica
Name: class, Length: 150, dtype: object
```

The Pandas Series is a one-dimensional labeled array that may hold any type of data (integer, string, float, python objects, etc.)

```
type(iris['class'])
```

```
pandas.core.series.Series
```

```
iris.dtypes
```

```
sepal_length    float64
sepal_width     float64
petal_length    float64
petal_width     float64
class            object
dtype: object
```

Tab completion for column names (as well as public attributes), `iris.<TAB>` , is enabled by default if you're using Jupyter.

For example, type `iris.` and then follow with the TAB key. Look for the `shape` attribute.

The `shape` attribute of pandas.DataFrame stores the number of rows and columns as a tuple (number of rows, number of columns).

```
iris.shape
```

```
(150, 5)
```

```
iris.info
```

```
<bound method DataFrame.info of      sepal_length  sepal_width  petal_length  petal_width
class
0             5.1          3.5           1.4          0.2     Iris-setosa
1             4.9          3.0           1.4          0.2     Iris-setosa
2             4.7          3.2           1.3          0.2     Iris-setosa
3             4.6          3.1           1.5          0.2     Iris-setosa
4             5.0          3.6           1.4          0.2     Iris-setosa
..            ...          ...           ...          ...             ...
145           6.7          3.0           5.2          2.3  Iris-virginica
146           6.3          2.5           5.0          1.9  Iris-virginica
147           6.5          3.0           5.2          2.0  Iris-virginica
148           6.2          3.4           5.4          2.3  Iris-virginica
149           5.9          3.0           5.1          1.8  Iris-virginica

[150 rows x 5 columns]>
```

```
import pandas as pd

values = {'dates':  ['20210305','20210316','20210328'],
          'status': ['Opened','Opened','Closed']
          }

demo = pd.DataFrame(values)
```

```
demo
```

	dates	status
0	20210305	Opened
1	20210316	Opened
2	20210328	Closed

```
demo['dates'] = pd.to_datetime(demo['dates'], format='%Y%m%d')
```

```
demo
```

	dates	status
0	2021-03-05	Opened
1	2021-03-16	Opened
2	2021-03-28	Closed

```
demo.to_csv('demo_df.csv')
```

3.2. Data Selection

We'll concentrate on how to slice, dice, and retrieve and set subsets of pandas objects in general. Because Series and DataFrame have received greater development attention in this area, they will be the key focus.

The axis labeling information in pandas objects is useful for a variety of reasons:

- Data is identified (metadata is provided) using established indicators, which is useful for analysis, visualization, and interactive console display.
- Allows for both implicit and explicit data alignment.
- Allows you to access and set subsets of the data set in an intuitive way.

Three different forms of multi-axis indexing are currently supported by pandas.

1. The indexing operators [] and attribute operator. in Python and NumPy offer quick and easy access to pandas data structures in a variety of situations.
2. `.loc` is mostly label-based, but it can also be used with a boolean array. .When the items are not found, .loc will produce a KeyError.
3. `.iloc` works with an integer array (from 0 to length-1 of the axis), but it can also work with a boolean array.

Except for slice indexers, which enable out-of-bounds indexing, .iloc will throw IndexError if a requested indexer is out-of-bounds. (This is in line with the Python/NumPy slice semantics.)

In this section we will use the Iris dataset. First we obtain the row and column names by using the following commands:

```
print(iris.index)
print(iris.columns)
```

```
RangeIndex(start=0, stop=150, step=1)
Index(['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'class'],
dtype='object')
```

```
iris.head()
```

	sepal_length	sepal_width	petal_length	petal_width	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

3.2.1. The indexing operators []

To begin, simply indicate the column and line (by using its index) you're interested in.

You can use the following command to get the sepal width of the fifth line (index is 4):

```
iris['sepal_width'][4]
```

```
3.6
```

```
# Not working
# iris[4]['sepal_width']
```

Note: Be careful, because this is not a matrix, and you might be tempted to insert the row first, then the column. Remember that it's a pandas DataFrame, and the [] operator operates on columns first, then the element of the pandas Series that results."

Sub-matrix retrieval is a simple procedure that requires only the specification of lists of indexes rather than scalars.

```
iris['sepal_width'][0:4]
```

```
0    3.5
1    3.0
2    3.2
3    3.1
Name: sepal_width, dtype: float64
```

```
iris[['petal_width','sepal_width']][0:4]
```

	petal_width	sepal_width
0	0.2	3.5
1	0.2	3.0
2	0.2	3.2
3	0.2	3.1

```
iris['sepal_width'][range(4)]
```

```
0    3.5
1    3.0
2    3.2
3    3.1
Name: sepal_width, dtype: float64
```

3.2.2. .loc()

You can use the `.loc()` method to get something similar to the other approach (as in a matrix) of obtaining data.

```
iris.loc[4,'sepal_width']
```

```
3.6
```

```
# rows at index labels between 0 and 4 (inclusive)
# See https://stackoverflow.com/questions/31593201/how-are-iloc-and-loc-different

iris.loc[0:4,'sepal_width']
```

```
0    3.5
1    3.0
2    3.2
3    3.1
4    3.6
Name: sepal_width, dtype: float64
```

```
iris.loc[range(4),['petal_width','sepal_width']]
```

	petal_width	sepal_width
0	0.2	3.5
1	0.2	3.0
2	0.2	3.2
3	0.2	3.1

3.2.3. .iloc()

Finally, there is `.iloc()`, which is a fully optimized function that defines the positions (as in a matrix). It requires you to define the cell using the row and column numbers.

```
iris.iloc[4,1]
```

```
3.6
```

The following commands produce the same output as `iris.loc[0:4,'sepal_width']` and `iris.loc[range(4),['petal_width','sepal_width']]`

```
# rows at index locations between 0 and 4 (exclusive)
# See https://stackoverflow.com/questions/31593201/how-are-iloc-and-loc-different

iris.iloc[0:4,1]
```

```
0    3.5
1    3.0
2    3.2
3    3.1
Name: sepal_width, dtype: float64
```

```
iris.iloc[range(4),[3,1]]
```

	petal_width	sepal_width
0	0.2	3.5
1	0.2	3.0
2	0.2	3.2
3	0.2	3.1

Note: .loc, .iloc, and also [] indexing can accept a callable as indexer as illustrated from the following examples. The callable must be a function with one argument (the calling Series or DataFrame) that returns valid output for indexing.

```
iris.loc[:,lambda df: ['petal_length','sepal_length']]
```

	petal_length	sepal_length
0	1.4	5.1
1	1.4	4.9
2	1.3	4.7
3	1.5	4.6
4	1.4	5.0
...	...	...
145	5.2	6.7
146	5.0	6.3
147	5.2	6.5
148	5.4	6.2
149	5.1	5.9

150 rows × 2 columns

```
iris.loc[lambda df: df['sepal_width'] > 3.5, :]
```

	sepal_length	sepal_width	petal_length	petal_width	class
4	5.0	3.6	1.4	0.2	Iris-setosa
5	5.4	3.9	1.7	0.4	Iris-setosa
10	5.4	3.7	1.5	0.2	Iris-setosa
14	5.8	4.0	1.2	0.2	Iris-setosa
15	5.7	4.4	1.5	0.4	Iris-setosa
16	5.4	3.9	1.3	0.4	Iris-setosa
18	5.7	3.8	1.7	0.3	Iris-setosa
19	5.1	3.8	1.5	0.3	Iris-setosa
21	5.1	3.7	1.5	0.4	Iris-setosa
22	4.6	3.6	1.0	0.2	Iris-setosa
32	5.2	4.1	1.5	0.1	Iris-setosa
33	5.5	4.2	1.4	0.2	Iris-setosa
44	5.1	3.8	1.9	0.4	Iris-setosa
46	5.1	3.8	1.6	0.2	Iris-setosa
48	5.3	3.7	1.5	0.2	Iris-setosa
109	7.2	3.6	6.1	2.5	Iris-virginica
117	7.7	3.8	6.7	2.2	Iris-virginica
131	7.9	3.8	6.4	2.0	Iris-virginica

3.3. Dealing with Problematic Data

3.3.1. Problem in setting index in pandas DataFrame

It may happen that the dataset contains an index column. How to import it correctly with Pandas?

We will use a very simple dataset, namely demo_df.csv (the file can be download from my github repository), that contains an index column (this is just a counter and not a feature).

```
import pandas as pd

# How to read CSV file from GitHub using pandas
# https://stackoverflow.com/questions/55240330/how-to-read-csv-file-from-github-using-pandas

# url = 'https://raw.githubusercontent.com/pairote-sat/SCMA248/main/demo_df'
url = 'https://raw.githubusercontent.com/pairote-sat/SCMA248/main/Data/demo_df.csv'

df1 = pd.read_csv(url)
print(df1.head())
df1.columns
```

```
---------------------------------------------------------------------------
SSLCertVerificationError                  Traceback (most recent call last)
/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/urllib/request.py in
do_open(self, http_class, req, **http_conn_args)
   1349                 h.request(req.get_method(), req.selector, req.data, headers,
-> 1350                           encode_chunked=req.has_header('Transfer-encoding'))
   1351             except OSError as err: # timeout error

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/http/client.py in
request(self, method, url, body, headers, encode_chunked)
   1261         """Send a complete request to the server."""
-> 1262         self._send_request(method, url, body, headers, encode_chunked)
   1263

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/http/client.py in
_send_request(self, method, url, body, headers, encode_chunked)
   1307             body = _encode(body, 'body')
-> 1308         self.endheaders(body, encode_chunked=encode_chunked)
   1309

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/http/client.py in
endheaders(self, message_body, encode_chunked)
   1256             raise CannotSendHeader()
-> 1257         self._send_output(message_body, encode_chunked=encode_chunked)
   1258

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/http/client.py in
_send_output(self, message_body, encode_chunked)
   1027         del self._buffer[:]
-> 1028         self.send(msg)
   1029

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/http/client.py in
send(self, data)
    967             if self.auto_open:
--> 968                 self.connect()
    969             else:

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/http/client.py in
connect(self)
   1431             self.sock = self._context.wrap_socket(self.sock,
-> 1432
server_hostname=server_hostname)
   1433

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/ssl.py in
wrap_socket(self, sock, server_side, do_handshake_on_connect, suppress_ragged_eofs,
server_hostname, session)
    422             context=self,
--> 423             session=session
    424         )

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/ssl.py in _create(cls,
sock, server_side, do_handshake_on_connect, suppress_ragged_eofs, server_hostname,
context, session)
    869                         raise ValueError("do_handshake_on_connect should not be
specified for non-blocking sockets")
--> 870                     self.do_handshake()
    871             except (OSError, ValueError):

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/ssl.py in
do_handshake(self, block)
   1138                 self.settimeout(None)
-> 1139             self._sslobj.do_handshake()
   1140         finally:

SSLCertVerificationError: [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed:
unable to get local issuer certificate (_ssl.c:1091)

During handling of the above exception, another exception occurred:

URLError                                  Traceback (most recent call last)
/var/folders/kl/h_r05n_j76n32kt0dwy7kynw0000gn/T/ipykernel_2309/3690297895.py in <module>
      7 url = 'https://raw.githubusercontent.com/pairote-
sat/SCMA248/main/Data/demo_df.csv'
      8
----> 9 df1 = pd.read_csv(url)
     10 print(df1.head())
     11 df1.columns

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-
packages/pandas/util/_decorators.py in wrapper(*args, **kwargs)
    309                     stacklevel=stacklevel,
    310                 )
--> 311             return func(*args, **kwargs)
    312
    313         return wrapper

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-
```

```
packages/pandas/io/parsers/readers.py in read_csv(filepath_or_buffer, sep, delimiter,
header, names, index_col, usecols, squeeze, prefix, mangle_dupe_cols, dtype, engine,
converters, true_values, false_values, skipinitialspace, skiprows, skipfooter, nrows,
na_values, keep_default_na, na_filter, verbose, skip_blank_lines, parse_dates,
infer_datetime_format, keep_date_col, date_parser, dayfirst, cache_dates, iterator,
chunksize, compression, thousands, decimal, lineterminator, quotechar, quoting,
doublequote, escapechar, comment, encoding, encoding_errors, dialect, error_bad_lines,
warn_bad_lines, on_bad_lines, delim_whitespace, low_memory, memory_map, float_precision,
storage_options)
    584     kwds.update(kwds_defaults)
    585
--> 586     return _read(filepath_or_buffer, kwds)
    587
    588

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-
packages/pandas/io/parsers/readers.py in _read(filepath_or_buffer, kwds)
    480
    481     # Create the parser.
--> 482     parser = TextFileReader(filepath_or_buffer, **kwds)
    483
    484     if chunksize or iterator:

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-
packages/pandas/io/parsers/readers.py in __init__(self, f, engine, **kwds)
    809             self.options["has_index_names"] = kwds["has_index_names"]
    810
--> 811         self._engine = self._make_engine(self.engine)
    812
    813     def close(self):

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-
packages/pandas/io/parsers/readers.py in _make_engine(self, engine)
   1038             )
   1039         # error: Too many arguments for "ParserBase"
-> 1040         return mapping[engine](self.f, **self.options)  # type: ignore[call-arg]
   1041
   1042     def _failover_to_python(self):

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-
packages/pandas/io/parsers/c_parser_wrapper.py in __init__(self, src, **kwds)
     49
     50         # open handles
---> 51         self._open_handles(src, kwds)
     52         assert self.handles is not None
     53

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-
packages/pandas/io/parsers/base_parser.py in _open_handles(self, src, kwds)
    227             memory_map=kwds.get("memory_map", False),
    228             storage_options=kwds.get("storage_options", None),
--> 229             errors=kwds.get("encoding_errors", "strict"),
    230         )
    231

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-
packages/pandas/io/common.py in get_handle(path_or_buf, mode, encoding, compression,
memory_map, is_text, errors, storage_options)
    612         compression=compression,
    613         mode=mode,
--> 614         storage_options=storage_options,
    615     )
    616

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-
packages/pandas/io/common.py in _get_filepath_or_buffer(filepath_or_buffer, encoding,
compression, mode, storage_options)
    310         # assuming storage_options is to be interpreted as headers
    311         req_info = urllib.request.Request(filepath_or_buffer,
headers=storage_options)
--> 312         with urlopen(req_info) as req:
    313             content_encoding = req.headers.get("Content-Encoding", None)
    314             if content_encoding == "gzip":

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-
packages/pandas/io/common.py in urlopen(*args, **kwargs)
    210     import urllib.request
    211
--> 212     return urllib.request.urlopen(*args, **kwargs)
    213
    214

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/urllib/request.py in
urlopen(url, data, timeout, cafile, capath, cadefault, context)
    220     else:
    221         opener = _opener
--> 222     return opener.open(url, data, timeout)
    223
    224 def install_opener(opener):
```

```
/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/urllib/request.py in
open(self, fullurl, data, timeout)
    523             req = meth(req)
    524
--> 525         response = self._open(req, data)
    526
    527         # post-process response

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/urllib/request.py in
_open(self, req, data)
    541         protocol = req.type
    542         result = self._call_chain(self.handle_open, protocol, protocol +
--> 543                                   '_open', req)
    544         if result:
    545             return result

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/urllib/request.py in
_call_chain(self, chain, kind, meth_name, *args)
    501         for handler in handlers:
    502             func = getattr(handler, meth_name)
--> 503             result = func(*args)
    504             if result is not None:
    505                 return result

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/urllib/request.py in
https_open(self, req)
   1391         def https_open(self, req):
   1392             return self.do_open(http.client.HTTPSConnection, req,
-> 1393                 context=self._context, check_hostname=self._check_hostname)
   1394
   1395         https_request = AbstractHTTPHandler.do_request_

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/urllib/request.py in
do_open(self, http_class, req, **http_conn_args)
   1350                           encode_chunked=req.has_header('Transfer-encoding'))
   1351             except OSError as err: # timeout error
-> 1352                 raise URLError(err)
   1353             r = h.getresponse()
   1354         except:

URLError: <urlopen error [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed:
unable to get local issuer certificate (_ssl.c:1091)>
```

```
## Uncomment these commands if the CSV dateset is stored locally.

# df1 = pd.read_csv('/Users/Kaemyuijang/SCMA248/demo_df.csv')
# print(df1.head())
# df1.columns
```

We want to specify that 'Unnamed: 0' is the index column while loading this data set with the following command (with the parameter `index_col`):

```
df1 = pd.read_csv(url, index_col = 0)
df1.head()
```

	dates	status
0	2021-03-05	Opened
1	2021-03-16	Opened
2	2021-03-28	Closed

The dataset is loaded and the index is correct after performing the command.

3.3.2. Convert Strings to Datetime

However, we see an issue right away: all of the data, including dates, has been parsed as integers (or, in other cases, as string). If the dates do not have a particularly unusual format, you can use the autodetection routines to identify the column that contains the date data. It works nicely with the following arguments when the data file is stored locally.

```
# df2 = pd.read_csv('/Users/Kaemyuijang/SCMA248/demo_df.csv', index_col = 0, parse_dates
= ['dates'])
# print(df2.head())
# df2.dtypes
```

For the same dataset downloaded from Github, if a column or index contains an unparseable date, the entire column or index will be returned unaltered as an object data type. For non-standard datetime parsing, use pd.to_datetime after pd.read_csv:

```
pd.to_datetime(df['DataFrame Column'], format=specify your format)
```

Remember that the date format for our example is yyyymmdd.

The following is a representation of this date format `format = '%d%m%Y'` (or `format = '%Y%m%d'`).

See https://datatofish.com/strings-to-datetime-pandas/ for more details.

```
df2 = pd.read_csv(url, index_col = 0, parse_dates = ['dates'])
print(df2)
df2.dtypes
```

```
       dates  status
0 2021-03-05  Opened
1 2021-03-16  Opened
2 2021-03-28  Closed
```

```
dates     datetime64[ns]
status            object
dtype: object
```

```
df2['dates'] = pd.to_datetime(df2['dates'], format='%d%m%Y')
print(df2)
df2.dtypes
```

```
       dates  status
0 2021-03-05  Opened
1 2021-03-16  Opened
2 2021-03-28  Closed
```

```
dates     datetime64[ns]
status            object
dtype: object
```

3.3.3. Missing values

We will concentrate on missing values, which is perhaps the most challenging data cleaning operation.

It's a good idea to have an overall sense of a data set before you start cleaning it. After that, you can develop a plan for cleaning the data.

To begin, I like to ask the following questions:

- What are the features?
- What sorts of data are required (int, float, text, boolean)?
- Is there any evident data missing (values that Pandas can detect)?
- Is there any other type of missing data that isn't as clear (and that Pandas can't easily detect)?

Let's have a look at an example by using a small sample data namely property_data.csv. The file can be obtained from Github: https://raw.githubusercontent.com/pairote-sat/SCMA248/main/property_data.csv.

In what follows, we also specify that 'PID' (personal indentifier) is the index column while loading this data set with the following command (with the parameter index_col):

```
url = 'https://raw.githubusercontent.com/pairote-
sat/SCMA248/main/Data/property_data.csv'

df = pd.read_csv(url, index_col = 0)
df
```

PID	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
10101.0	104.0	Khao San	Y	3	1	1000
10102.0	197.0	Silom	N	3	1.5	--
10103.0	NaN	Silom	N	NaN	1	850
10104.0	201.0	Sukhumvit	12	1	NaN	700
NaN	203.0	Sukhumvit	Y	3	2	1600
10106.0	207.0	Sukhumvit	Y	NaN	1	800
10107.0	NaN	Thonglor	NaN	2	HURLEY	950
10108.0	213.0	Rama 1	Y	--	1	NaN
10109.0	215.0	Rama 1	Y	na	2	1800

We notice that the PID (personal identifiers) as the index name has a missing value, i.e. NaN (not any number). We will replace this missing PID with 10105 and also convert from floats to integers.

```
rowindex = df.index.tolist()
rowindex[4] = 10105.0
rowindex = [int(i) for i in rowindex]

df.index = rowindex

print(df.loc[:,'ST_NUM'])
```

```
10101    104.0
10102    197.0
10103      NaN
10104    201.0
10105    203.0
10106    207.0
10107      NaN
10108    213.0
10109    215.0
Name: ST_NUM, dtype: float64
```

Alternatively, one can use Numpy to produce the same result. Simply run the following commands. Here we use `.astype()` method to convert the type of an array.

```
df = pd.read_csv(url, index_col = 0)
df

import numpy as np
rowindex = df.index.to_numpy()

rowindex[4] = 10105.0

df.index = rowindex.astype(int)

print(df.loc[:,'ST_NUM'])
```

```
10101    104.0
10102    197.0
10103      NaN
10104    201.0
10105    203.0
10106    207.0
10107      NaN
10108    213.0
10109    215.0
Name: ST_NUM, dtype: float64
```

Now I can answer my first question: what are features? The following features can be obtained from the column names:

- ST_NUM is the street number
- ST_NAME is the street name
- OWN_OCCUPIED: Is the residence owner occupied?
- NUM_BEDROOMS: the number of rooms

We can also respond to the question, What are the expected types?

- ST_NUM is either a float or an int… a numeric type of some sort
- ST_NAME is a string variable.
- OWN_OCCUPIED: string; OWN_OCCUPIED: string; OWN _OCCUPIED N (“No”) or Y (“Yes”)
- NUM_BEDROOMS is a numeric type that can be either float or int.

3.3.4. Standard Missing Values

So, what exactly do I mean when I say “standard missing values?” These are missing values that Pandas can detect.

Let’s return to our initial dataset and examine the “Street Number” column.

There are an empty cell in the third row (from the original file). A value of “NaN” appears in the seventh row.

Both of these numbers are obviously missing. Let’s see how Pandas handle these situations. We can see that Pandas filled in the blank space with “NaN”.

We can confirm that both the missing value and “NA” were detected as missing values using the isnull() method. True for both boolean responses.

```
df['ST_NUM'].isnull()
```

```
10101    False
10102    False
10103     True
10104    False
10105    False
10106    False
10107     True
10108    False
10109    False
Name: ST_NUM, dtype: bool
```

Similarly, for the NUM_BEDROOMS column of the original CSV file, users manually entering missing values with different names “n/a” and “NA”. Pandas also recognized these as missing values and filled with “NaN”.

```
df['NUM_BEDROOMS']
```

```
10101      3
10102      3
10103    NaN
10104      1
10105      3
10106    NaN
10107      2
10108     --
10109     na
Name: NUM_BEDROOMS, dtype: object
```

3.3.5. Missing Values That Aren’t Standard

It is possible that there are missing values with different formats in some cases.

There are two other missing values in this column of different formats

- na
- --

Putting this different format in a list is a simple approach to detect them. When we import the data, Pandas will immediately recognize them. Here’s an example of how we might go about it.

```
# Making a list of missing value types
missing_values = ["na", "--"]

df = pd.read_csv(url, index_col = 0, na_values = missing_values)

df
```

PID	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
10101.0	104.0	Khao San	Y	3.0	1	1000.0
10102.0	197.0	Silom	N	3.0	1.5	NaN
10103.0	NaN	Silom	N	NaN	1	850.0
10104.0	201.0	Sukhumvit	12	1.0	NaN	700.0
NaN	203.0	Sukhumvit	Y	3.0	2	1600.0
10106.0	207.0	Sukhumvit	Y	NaN	1	800.0
10107.0	NaN	Thonglor	NaN	2.0	HURLEY	950.0
10108.0	213.0	Rama 1	Y	NaN	1	NaN
10109.0	215.0	Rama 1	Y	NaN	2	1800.0

3.3.6. Unexpected Missing Values

We have observed both standard and non-standard missing data so far. What if we have a type that is not expected?

For instance, if our feature is supposed to be a string but it's a numeric type, it's technically a missing value.

Take a look at the column labeled "OWN_OCCUPIED" to understand what I'm talking about.

```
df['OWN_OCCUPIED']
```

```
PID
10101.0      Y
10102.0      N
10103.0      N
10104.0     12
NaN          Y
10106.0      Y
10107.0    NaN
10108.0      Y
10109.0      Y
Name: OWN_OCCUPIED, dtype: object
```

We know Pandas will recognize the empty cell in row seven as a missing value because of our prior examples.

The number 12 appears in the fourth row. This number type should be a missing value because the result for Owner Occupied should clearly be a string (Y or N). Because this example is a little more complicated, we will need to find a plan for identifying missing values. There are a few alternative routes to take, but this is how I'm going to go about it.

1. Loop through The OWN OCCUPIED column.
2. Convert the entry to an integer.
3. If the entry may be transformed to an integer, enter a missing value.
4. We know the number cannot be an integer if it cannott be an integer.

```
df = pd.read_csv(url, index_col = 0)
df

import numpy as np
rowindex = df.index.to_numpy()

rowindex[4] = 10105.0

df.index = rowindex.astype(int)
```

```
# Detecting numbers
cnt=10101
for row in df['OWN_OCCUPIED']:
    try:
        int(row)
        df.loc[cnt, 'OWN_OCCUPIED']=np.nan
    except ValueError:
        pass
    cnt+=1
```

```
df['OWN_OCCUPIED']
```

```
10101      Y
10102      N
10103      N
10104    NaN
10105      Y
10106      Y
10107    NaN
10108      Y
10109      Y
Name: OWN_OCCUPIED, dtype: object
```

In the code, we loop through each entry in the “Owner Occupied” column. To try to change the entry to an integer, we use int(row). If the value can be changed to an integer, we change the entry to a missing value using np.nan from Numpy. On the other hand, if the value cannot be changed to an integer, we pass it and continue.

You will notice that I have used try and except ValueError. This is called exception handling, and we use this to handle errors.

If we tried to change an entry to an integer and it could not be changed, a ValueError would be returned and the code would terminate. To deal with this, we use exception handling to detect these errors and continue.

3.4. Data Manipulation

We have learned how to select the data we want, we will need to learn how to manipulate it. Using aggregation methods to work with columns or rows is one of the most straightforward things we can perform.

All of these functions always **return a number** when applied to a row or column.

We can specify whether the function should be applied to

- the rows for each column using the `axis=0` keyword on the function argument, or
- the columns for each row using the `axis=1` keyword on the function argument.

Function	Description
df.describe()	Returns a summary statistics of numerical column
df.mean()	Returns the average of all columns in a dataset
df.corr()	Returns the correlation between columns in a DataFrame
df.count()	Returns the number of non-null values in each DataFrame column
df.max()	Returns the highest value in each column
df.min()	Returns the lowest value in each column
df.median()	Returns the median in each column
df.std()	Returns the standard deviation in each column

```
iris = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data', header=None,
                 names = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width',
'class'])

type(iris)
```

```
pandas.core.frame.DataFrame
```

```
iris.describe()
```

	sepal_length	sepal_width	petal_length	petal_width
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667
std	0.828066	0.433594	1.764420	0.763161
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

You have the number of observations, their average value, standard deviation, minimum and maximum values, and certain percentiles for all numerical features" (25 percent, 50 percent, and 75 percent). This offers you a fair picture of how each feature is distributed.

The following command illustrate how to use the `max()` function.

```
iris.max(axis = 0)
```

```
sepal_length               7.9
sepal_width                4.4
petal_length               6.9
petal_width                2.5
class           Iris-virginica
dtype: object
```

We can perform operations on all values in rows, columns, or a subset of both.

The following example shows how to find the standardized values of each column of the Iris dataset. We need to firstly drop the class column, which is a categorical variable using `drop()` function with `axis = 1`.

Recall that the Z-scores and standardized values (sometimes known as standard scores or normal deviates) are the same thing. When you take a data point and scale it by population data, you get a standardized value. It informs us how distant we are from the mean in terms of standard deviations.

```
iris_drop = iris.drop('class', axis = 1)
```

```
print(iris_drop.mean())
print(iris_drop.std())
```

```
sepal_length    5.843333
sepal_width     3.054000
petal_length    3.758667
petal_width     1.198667
dtype: float64
sepal_length    0.828066
sepal_width     0.433594
petal_length    1.764420
petal_width     0.763161
dtype: float64
```

```
def z_score_standardization(series):
    return (series - series.mean()) / series.std(ddof = 1)

#iris_normalized = iris_drop
#for col in iris_normalized.columns:
#    iris_normalized[col] = z_score_standardization(iris_normalized[col])

iris_normalized = {}
for col in iris_drop.columns:
    iris_normalized[col] = z_score_standardization(iris_drop[col])

iris_normalized = pd.DataFrame(iris_normalized)
```

```
iris_normalized.head()
```

	sepal_length	sepal_width	petal_length	petal_width
0	-0.897674	1.028611	-1.336794	-1.308593
1	-1.139200	-0.124540	-1.336794	-1.308593
2	-1.380727	0.336720	-1.393470	-1.308593
3	-1.501490	0.106090	-1.280118	-1.308593
4	-1.018437	1.259242	-1.336794	-1.308593

Alternatively, the following commands produce the same result.

```
del(iris_normalized)

iris_normalized = (iris_drop - iris_drop.mean())/iris_drop.std(ddof=1)
iris_normalized.head()
```

	sepal_length	sepal_width	petal_length	petal_width
0	-0.897674	1.028611	-1.336794	-1.308593
1	-1.139200	-0.124540	-1.336794	-1.308593
2	-1.380727	0.336720	-1.393470	-1.308593
3	-1.501490	0.106090	-1.280118	-1.308593
4	-1.018437	1.259242	-1.336794	-1.308593

Alternatively, we can standardize a Pandas Column with Z-Score Scaling using scikit-learn.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(iris_drop)

iris_scaled = scaler.fit_transform(iris_drop)
iris_scaled = pd.DataFrame(iris_scaled, columns=iris_drop.columns)
print(iris_scaled)
```

```
     sepal_length  sepal_width  petal_length  petal_width
0       -0.900681     1.032057     -1.341272    -1.312977
1       -1.143017    -0.124958     -1.341272    -1.312977
2       -1.385353     0.337848     -1.398138    -1.312977
3       -1.506521     0.106445     -1.284407    -1.312977
4       -1.021849     1.263460     -1.341272    -1.312977
..            ...          ...           ...          ...
145      1.038005    -0.124958      0.819624     1.447956
146      0.553333    -1.281972      0.705893     0.922064
147      0.795669    -0.124958      0.819624     1.053537
148      0.432165     0.800654      0.933356     1.447956
149      0.068662    -0.124958      0.762759     0.790591

[150 rows x 4 columns]
```

Note scikit-learn uses np.std which by **default is the population standard deviation** (where the sum of squared deviations are divided by the number of observations), while the sample standard deviations (where the denominator is number of observations - 1) are used by pandas. This is a correction factor determined by the degrees of freedom in order to obtain an unbiased estimate of the population standard deviation (ddof). Numpy and scikit-learn calculations utilize ddof=0 by default, whereas pandas uses ddof=1 (docs).

As a result, the above results are different.

https://stackoverflow.com/questions/44220290/sklearn-standardscaler-result-different-to-manual-result

In the next example, we simply can apply any binary arithmetical operation (+,-,*,/) to an entire dataframe.

```
iris_drop**2
```

	sepal_length	sepal_width	petal_length	petal_width
0	26.01	12.25	1.96	0.04
1	24.01	9.00	1.96	0.04
2	22.09	10.24	1.69	0.04
3	21.16	9.61	2.25	0.04
4	25.00	12.96	1.96	0.04
...	...	...	...	...
145	44.89	9.00	27.04	5.29
146	39.69	6.25	25.00	3.61
147	42.25	9.00	27.04	4.00
148	38.44	11.56	29.16	5.29
149	34.81	9.00	26.01	3.24

150 rows × 4 columns

Any function can be applied to a DataFrame or Series by passing its name as an argument to the `apply` method. For example, in the following code, we use the NumPy library's `floor` function to return the floor of the input, element-wise of each value in the DataFrame.

```
import numpy as np

iris_drop.apply(np.floor)
```

	sepal_length	sepal_width	petal_length	petal_width
0	5.0	3.0	1.0	0.0
1	4.0	3.0	1.0	0.0
2	4.0	3.0	1.0	0.0
3	4.0	3.0	1.0	0.0
4	5.0	3.0	1.0	0.0
...	...	...	...	...
145	6.0	3.0	5.0	2.0
146	6.0	2.0	5.0	1.0
147	6.0	3.0	5.0	2.0
148	6.0	3.0	5.0	2.0
149	5.0	3.0	5.0	1.0

150 rows × 4 columns

If we need to design a specific function to apply it, we can write an in-line function, commonly known as a λ-function. A λ-function is a function without a name.

It is only necessary to specify the parameters it receives, between the lambda keyword and the colon (:).

In the next example, only one parameter is needed, which will be the value of each element in the DataFrame.

```
iris_drop.apply(lambda x: np.log10(x))
```

	sepal_length	sepal_width	petal_length	petal_width
0	0.707570	0.544068	0.146128	-0.698970
1	0.690196	0.477121	0.146128	-0.698970
2	0.672098	0.505150	0.113943	-0.698970
3	0.662758	0.491362	0.176091	-0.698970
4	0.698970	0.556303	0.146128	-0.698970
...	...	...	...	...
145	0.826075	0.477121	0.716003	0.361728
146	0.799341	0.397940	0.698970	0.278754
147	0.812913	0.477121	0.716003	0.301030
148	0.792392	0.531479	0.732394	0.361728
149	0.770852	0.477121	0.707570	0.255273

150 rows × 4 columns

3.4.1. Add A New Column To An Existing Pandas DataFrame

Adding new values in our DataFrame is another simple manipulation technique. This can be done directly over a DataFrame with the assign operator (=).

For example, we can assign a Series to a selection of a column that does not exist to add a new column to a DataFrame.

You should be aware that previous values will be overridden if a column with the same name already exists.

In the following example, we create a new column entitled sepal_length_normalized by adding the standardized values of the sepal_length column.

```
iris['sepal_length_normalized'] = (iris['sepal_length'] - iris['sepal_length'].mean()) /
iris['sepal_length'].std()
iris.head()
```

	sepal_length	sepal_width	petal_length	petal_width	class	sepal_length_normalized
0	5.1	3.5	1.4	0.2	Iris-setosa	-0.897674
1	4.9	3.0	1.4	0.2	Iris-setosa	-1.139200
2	4.7	3.2	1.3	0.2	Iris-setosa	-1.380727
3	4.6	3.1	1.5	0.2	Iris-setosa	-1.501490
4	5.0	3.6	1.4	0.2	Iris-setosa	-1.018437

```
iris['sepal_length_normalized'] = (iris['sepal_length'] - iris['sepal_length'].mean()) /
iris['sepal_length'].std(ddof=0)
iris.head()
```

	sepal_length	sepal_width	petal_length	petal_width	class	sepal_length_normalized
0	5.1	3.5	1.4	0.2	Iris-setosa	-0.900681
1	4.9	3.0	1.4	0.2	Iris-setosa	-1.143017
2	4.7	3.2	1.3	0.2	Iris-setosa	-1.385353
3	4.6	3.1	1.5	0.2	Iris-setosa	-1.506521
4	5.0	3.6	1.4	0.2	Iris-setosa	-1.021849

Alternatively, we can use `concat` function to add a Series (s) to a Pandas DataFrame (df) as a new column with an arguement `axis = 1`. The name of the new column can be set by using `Series.rename` as in the following command:

`df = pd.concat((df, s.rename('CoolColumnName')), axis=1)`

https://stackoverflow.com/questions/39047915/concat-series-onto-dataframe-with-column-name

```
pd.concat([iris, (iris['sepal_length']-1).rename('new_name') ], axis = 1)
```

	sepal_length	sepal_width	petal_length	petal_width	class	sepal_length_normalized	new_name
0	5.1	3.5	1.4	0.2	Iris-setosa	-0.900681	4.1
1	4.9	3.0	1.4	0.2	Iris-setosa	-1.143017	3.9
2	4.7	3.2	1.3	0.2	Iris-setosa	-1.385353	3.7
3	4.6	3.1	1.5	0.2	Iris-setosa	-1.506521	3.6
4	5.0	3.6	1.4	0.2	Iris-setosa	-1.021849	4.0
...	...	...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	Iris-virginica	1.038005	5.7
146	6.3	2.5	5.0	1.9	Iris-virginica	0.553333	5.3
147	6.5	3.0	5.2	2.0	Iris-virginica	0.795669	5.5
148	6.2	3.4	5.4	2.3	Iris-virginica	0.432165	5.2
149	5.9	3.0	5.1	1.8	Iris-virginica	0.068662	4.9

150 rows × 7 columns

We can now use the `drop` function to delete a column from the DataFrame; this removes the indicated rows if axis=0, or the indicated columns if axis=1.

All Pandas functions that change the contents of a DataFrame, such as the drop function, return a duplicate of the updated data rather than overwriting the DataFrame. As a result, the original DataFrame is preserved. Set the keyword `inplace` to `True` if you do not wish to maintain the old settings. This keyword is set to `False` by default, which means that a copy of the data is returned.

The following commands remove the column, namely 'sepal_length_normalized', we have just added to the Iris dataset.

```
iris.columns
```

```
Index(['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'class',
       'sepal_length_normalized'],
      dtype='object')
```

```
print(iris.drop('sepal_length_normalized', axis = 1).head())

print(iris.head())
```

```
   sepal_length  sepal_width  petal_length  petal_width        class
0           5.1          3.5           1.4          0.2  Iris-setosa
1           4.9          3.0           1.4          0.2  Iris-setosa
2           4.7          3.2           1.3          0.2  Iris-setosa
3           4.6          3.1           1.5          0.2  Iris-setosa
4           5.0          3.6           1.4          0.2  Iris-setosa
   sepal_length  sepal_width  petal_length  petal_width        class  \
0           5.1          3.5           1.4          0.2  Iris-setosa
1           4.9          3.0           1.4          0.2  Iris-setosa
2           4.7          3.2           1.3          0.2  Iris-setosa
3           4.6          3.1           1.5          0.2  Iris-setosa
4           5.0          3.6           1.4          0.2  Iris-setosa

   sepal_length_normalized
0                -0.900681
1                -1.143017
2                -1.385353
3                -1.506521
4                -1.021849
```

```
iris.drop('sepal_length_normalized', axis = 1, inplace = True)

print(iris.head())
```

```
   sepal_length  sepal_width  petal_length  petal_width        class
0           5.1          3.5           1.4          0.2  Iris-setosa
1           4.9          3.0           1.4          0.2  Iris-setosa
2           4.7          3.2           1.3          0.2  Iris-setosa
3           4.6          3.1           1.5          0.2  Iris-setosa
4           5.0          3.6           1.4          0.2  Iris-setosa
```

3.4.2. Appending a row to a dataframe and specify its index label

In this section, we will learn how to add/remove new rows and remove missing values. We will use the dataset, namely property_data.csv.

A general solution when appending a row to a dataframe and specify its index label is to create the row, transform the new row data into a pandas series, name it to the index you want to have and then append it to the data frame. Don't forget to overwrite the original data frame with the one with appended row. The following commands produce the required result.

See https://stackoverflow.com/questions/16824607/pandas-appending-a-row-to-a-dataframe-and-specify-its-index-label for more details.

See also https://www.geeksforgeeks.org/python-pandas-dataframe-append/ and https://thispointer.com/python-pandas-how-to-add-rows-in-a-dataframe-using-dataframe-append-loc-iloc/#3 .

```
import pandas as pd

filepath = '/Users/Kaemyuijang/SCMA248/Data/property_data.csv'

df = pd.read_csv(filepath, index_col = 0)
df.head()
```

PID	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
10101.0	104.0	Khao San	Y	3	1	1000
10102.0	197.0	Silom	N	3	1.5	--
10103.0	NaN	Silom	N	NaN	1	850
10104.0	201.0	Sukhumvit	12	1	NaN	700
NaN	203.0	Sukhumvit	Y	3	2	1600

```
url = 'https://raw.githubusercontent.com/pairote-
sat/SCMA248/main/Data/property_data.csv'

df = pd.read_csv(url, index_col = 0)
df.head()
```

PID	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
10101.0	104.0	Khao San	Y	3	1	1000
10102.0	197.0	Silom	N	3	1.5	--
10103.0	NaN	Silom	N	NaN	1	850
10104.0	201.0	Sukhumvit	12	1	NaN	700
NaN	203.0	Sukhumvit	Y	3	2	1600

```
df.index
```

```
Float64Index([10101.0, 10102.0, 10103.0, 10104.0, nan, 10106.0, 10107.0,
              10108.0, 10109.0],
             dtype='float64', name='PID')
```

```
#new_observation = {'ST_NUM': 555 , 'OWN_OCCUPIED': 'Y', 'NUM_BEDROOMS': 2,'NUM_BATH':
1,'SQ_FT': 200}

new_observation = pd.Series({'ST_NUM': 555 , 'OWN_OCCUPIED': 'Y', 'NUM_BEDROOMS':
2,'NUM_BATH': 1,'SQ_FT': 200}, name = 10110.0)
```

```
new_observation
```

```
ST_NUM          555
OWN_OCCUPIED      Y
NUM_BEDROOMS      2
NUM_BATH          1
SQ_FT           200
Name: 10110.0, dtype: object
```

```
df.append(new_observation)
```

PID	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
10101.0	104.0	Khao San	Y	3	1	1000
10102.0	197.0	Silom	N	3	1.5	--
10103.0	NaN	Silom	N	NaN	1	850
10104.0	201.0	Sukhumvit	12	1	NaN	700
NaN	203.0	Sukhumvit	Y	3	2	1600
10106.0	207.0	Sukhumvit	Y	NaN	1	800
10107.0	NaN	Thonglor	NaN	2	HURLEY	950
10108.0	213.0	Rama 1	Y	--	1	NaN
10109.0	215.0	Rama 1	Y	na	2	1800
10110.0	555.0	NaN	Y	2	1	200

Note In case that, we do not define a name for our pandas series, i.e. we simply define `next_observation = pd.Series({'ST_NUM': 999 , 'OWN_OCCUPIED': 'Y', 'NUM_BEDROOMS': 2,'NUM_BATH': 1,'SQ_FT': 200})` without the flag `name = 10110.0` as an argument. We must set the `ignore_index` flag in the append method to `True`, otherwise the commands will produce an error as follows:

```
next_observation = pd.Series({'ST_NUM': 999 , 'OWN_OCCUPIED': 'N', 'NUM_BEDROOMS':
5,'NUM_BATH': 4,'SQ_FT': 3500})
```

```
print(next_observation)
print(df)
```

```
ST_NUM           999
OWN_OCCUPIED       N
NUM_BEDROOMS       5
NUM_BATH           4
SQ_FT           3500
dtype: object
         ST_NUM    ST_NAME OWN_OCCUPIED NUM_BEDROOMS NUM_BATH SQ_FT
PID
10101.0   104.0   Khao San            Y            3        1  1000
10102.0   197.0      Silom            N            3      1.5    --
10103.0     NaN      Silom            N          NaN        1   850
10104.0   201.0  Sukhumvit           12            1      NaN   700
NaN       203.0  Sukhumvit            Y            3        2  1600
10106.0   207.0  Sukhumvit            Y          NaN        1   800
10107.0     NaN   Thonglor          NaN            2   HURLEY   950
10108.0   213.0     Rama 1            Y           --        1   NaN
10109.0   215.0     Rama 1            Y           na        2  1800
```

```
# Without setting the ignore_index flag in the append method to True, this produces an
eror.

df.append(next_observation)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/var/folders/kl/h_r05n_j76n32kt0dwy7kynw0000gn/T/ipykernel_1214/4163196459.py in <module>
      1 # Without setting the ignore_index flag in the append method to True, this
produces an eror.
      2
----> 3 df.append(next_observation)

~/opt/anaconda3/lib/python3.7/site-packages/pandas/core/frame.py in append(self, other,
ignore_index, verify_integrity, sort)
   8933             if other.name is None and not ignore_index:
   8934                 raise TypeError(
-> 8935                     "Can only append a Series if ignore_index=True "
   8936                     "or if the Series has a name"
   8937                 )

TypeError: Can only append a Series if ignore_index=True or if the Series has a name
```

```
# Setting the ignore_index flag in the append method to True

df.append(next_observation,ignore_index=True)
```

	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
0	104.0	Khao San	Y	3	1	1000
1	197.0	Silom	N	3	1.5	--
2	NaN	Silom	N	NaN	1	850
3	201.0	Sukhumvit	12	1	NaN	700
4	203.0	Sukhumvit	Y	3	2	1600
5	207.0	Sukhumvit	Y	NaN	1	800
6	NaN	Thonglor	NaN	2	HURLEY	950
7	213.0	Rama 1	Y	--	1	NaN
8	215.0	Rama 1	Y	na	2	1800
9	999.0	NaN	N	5	4	3500

Note The resulting (new) DataFrame's index is not same as original dataframe because ignore_index is passed as True in the `append()` function.

The next complete example illustrates how to add multiple rows to dataframe.

```
print(new_observation)
print(next_observation)
```

```
ST_NUM          555
OWN_OCCUPIED      Y
NUM_BEDROOMS      2
NUM_BATH          1
SQ_FT           200
Name: 10110.0, dtype: object
ST_NUM           999
OWN_OCCUPIED       N
NUM_BEDROOMS       5
NUM_BATH           4
SQ_FT           3500
dtype: object
```

```
listOfSeries = [new_observation,next_observation]

new_df = df.append(listOfSeries, ignore_index = True)
```

Finally, to remove the row(s), we can apply the drop method once more. Now we must set the axis to 0 and specify the row index we want to remove.

```
new_df.drop([9,10], axis = 0, inplace = True)
```

```
new_df
```

	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
0	104.0	Khao San	Y	3	1	1000
1	197.0	Silom	N	3	1.5	--
2	NaN	Silom	N	NaN	1	850
3	201.0	Sukhumvit	12	1	NaN	700
4	203.0	Sukhumvit	Y	3	2	1600
5	207.0	Sukhumvit	Y	NaN	1	800
6	NaN	Thonglor	NaN	2	HURLEY	950
7	213.0	Rama 1	Y	--	1	NaN
8	215.0	Rama 1	Y	na	2	1800

By applying the `drop()` function to the result of the `isnull()` method, missing values can be removed. This has the same effect as filtering the NaN values, as explained above, but instead of returning a view, a duplicate of the DataFrame minus the NaN values is returned.

```
index = new_df.index[new_df['ST_NUM'].isnull()]

# Altenatively, one can obtain the index
new_index = new_df['ST_NUM'].index[new_df['ST_NUM'].apply(np.isnan)]

print(new_df.drop(index, axis = 0))
print(new_df.drop(new_index, axis = 0))
```

```
   ST_NUM    ST_NAME OWN_OCCUPIED NUM_BEDROOMS NUM_BATH SQ_FT
0   104.0   Khao San            Y            3        1  1000
1   197.0      Silom            N            3      1.5    --
3   201.0  Sukhumvit           12            1      NaN   700
4   203.0  Sukhumvit            Y            3        2  1600
5   207.0  Sukhumvit            Y          NaN        1   800
7   213.0     Rama 1            Y           --        1   NaN
8   215.0     Rama 1            Y           na        2  1800
   ST_NUM    ST_NAME OWN_OCCUPIED NUM_BEDROOMS NUM_BATH SQ_FT
0   104.0   Khao San            Y            3        1  1000
1   197.0      Silom            N            3      1.5    --
3   201.0  Sukhumvit           12            1      NaN   700
4   203.0  Sukhumvit            Y            3        2  1600
5   207.0  Sukhumvit            Y          NaN        1   800
7   213.0     Rama 1            Y           --        1   NaN
8   215.0     Rama 1            Y           na        2  1800
```

Instead of using the generic drop function, we can use the particular `dropna()` function to remove NaN values. We must set the `how` keyword to `any` if we wish to delete any record that includes a NaN value. We can use the `subset` keyword to limit it to a specific subset of columns. The effect will be the same as if we used the drop function, as shown below:

Note The parameter `how{'any', 'all'}`, default 'any' determine if row or column is removed from DataFrame, when we have at least one NA or all NA.

- 'any' : If any NA values are present, drop that row or column.
- 'all' : If all values are NA, drop that row or column.

```
df.dropna(how = 'any', subset = ['ST_NUM'])
```

PID	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
10101.0	104.0	Khao San	Y	3	1	1000
10102.0	197.0	Silom	N	3	1.5	--
10104.0	201.0	Sukhumvit	12	1	NaN	700
NaN	203.0	Sukhumvit	Y	3	2	1600
10106.0	207.0	Sukhumvit	Y	NaN	1	800
10108.0	213.0	Rama 1	Y	--	1	NaN
10109.0	215.0	Rama 1	Y	na	2	1800

```
# Setting subset keyword with a subset of columns

df.dropna(how = 'any', subset = ['ST_NUM','NUM_BATH'])
```

PID	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
10101.0	104.0	Khao San	Y	3	1	1000
10102.0	197.0	Silom	N	3	1.5	--
NaN	203.0	Sukhumvit	Y	3	2	1600
10106.0	207.0	Sukhumvit	Y	NaN	1	800
10108.0	213.0	Rama 1	Y	--	1	NaN
10109.0	215.0	Rama 1	Y	na	2	1800

If we wish to fill the rows containing NaN with another value instead of removing them, we may use the `fillna()` method and specify the value to use. If we only want to fill certain columns, we must pass a dictionary as an argument to the `fillna()` function, with the names of the columns as the key and the character to use for filling as the value.

```
df.fillna(value = {'ST_NUM':0})
```

PID	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
10101.0	104.0	Khao San	Y	3	1	1000
10102.0	197.0	Silom	N	3	1.5	--
10103.0	0.0	Silom	N	NaN	1	850
10104.0	201.0	Sukhumvit	12	1	NaN	700
NaN	203.0	Sukhumvit	Y	3	2	1600
10106.0	207.0	Sukhumvit	Y	NaN	1	800
10107.0	0.0	Thonglor	NaN	2	HURLEY	950
10108.0	213.0	Rama 1	Y	--	1	NaN
10109.0	215.0	Rama 1	Y	na	2	1800

```
df.fillna(value = {'ST_NUM': -1, 'NUM_BEDROOMS': -1 })
```

PID	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
10101.0	104.0	Khao San	Y	3	1	1000
10102.0	197.0	Silom	N	3	1.5	--
10103.0	-1.0	Silom	N	-1	1	850
10104.0	201.0	Sukhumvit	12	1	NaN	700
NaN	203.0	Sukhumvit	Y	3	2	1600
10106.0	207.0	Sukhumvit	Y	-1	1	800
10107.0	-1.0	Thonglor	NaN	2	HURLEY	950
10108.0	213.0	Rama 1	Y	--	1	NaN
10109.0	215.0	Rama 1	Y	na	2	1800

3.4.3. Sorting

Sorting by columns is another important feature we will need while examining at our data. Using the sort method, we can sort a DataFrame by any column. We only need to execute the following commands to see the first five rows of data sorted in descending order (i.e., from the largest to the lowest values) and using the Value column:

Here we will work with the Iris dataset.

```
print(iris.columns)
```

```
Index(['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'class'],
dtype='object')
```

```
iris.sort_values(by = ['sepal_length'], ascending = False)
```

	sepal_length	sepal_width	petal_length	petal_width	class
131	7.9	3.8	6.4	2.0	Iris-virginica
135	7.7	3.0	6.1	2.3	Iris-virginica
122	7.7	2.8	6.7	2.0	Iris-virginica
117	7.7	3.8	6.7	2.2	Iris-virginica
118	7.7	2.6	6.9	2.3	Iris-virginica
...	...	...	...	...	...
41	4.5	2.3	1.3	0.3	Iris-setosa
42	4.4	3.2	1.3	0.2	Iris-setosa
38	4.4	3.0	1.3	0.2	Iris-setosa
8	4.4	2.9	1.4	0.2	Iris-setosa
13	4.3	3.0	1.1	0.1	Iris-setosa

150 rows × 5 columns

The following command sorts the sepal_lenghth column followed by the petal_length column.

```
iris.sort_values(by = ['sepal_length','petal_length'], ascending = False)
```

	sepal_length	sepal_width	petal_length	petal_width	class
131	7.9	3.8	6.4	2.0	Iris-virginica
118	7.7	2.6	6.9	2.3	Iris-virginica
117	7.7	3.8	6.7	2.2	Iris-virginica
122	7.7	2.8	6.7	2.0	Iris-virginica
135	7.7	3.0	6.1	2.3	Iris-virginica
...	...	...	...	...	...
41	4.5	2.3	1.3	0.3	Iris-setosa
8	4.4	2.9	1.4	0.2	Iris-setosa
38	4.4	3.0	1.3	0.2	Iris-setosa
42	4.4	3.2	1.3	0.2	Iris-setosa
13	4.3	3.0	1.1	0.1	Iris-setosa

150 rows × 5 columns

Note

1. The `inplace` keyword indicates that the DataFrame will be overwritten, hence no new DataFrame will be returned.
2. When `ascending = True` is used instead of `ascending = False`, the values are sorted in ascending order (i.e., from the smallest to the largest values).
3. If we wish to restore the original order, we can use the `sort_index` method and sort by an index.

```
# restore the original order

iris.sort_index(axis = 0, ascending = True, inplace = True)
iris.head()
```

	sepal_length	sepal_width	petal_length	petal_width	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

```
# Importing sales dataset for the next section

filepath = '/Users/Kaemyuijang/SCMA248/Data/SalesData.csv'

# pd.read_csv(filepath, header = None, skiprows = 1, names =
['Trans_no','Name','Date','Product','Units','Dollars','Location'])

# header = 0 means that the first row to use as the column names
sales = pd.read_csv(filepath, header = 0,  index_col = 0)

# https://stackoverflow.com/questions/25015711/time-data-does-not-match-format
sales['Date'] =  pd.to_datetime(sales['Date'], format='%d/%m/%Y')

sales.head()
```

Trans_Number	Name	Date	Product	Units	Dollars	Location
1	Betsy	2004-04-01	lip gloss	45	137.20	south
2	Hallagan	2004-03-10	foundation	50	152.01	midwest
3	Ashley	2005-02-25	lipstick	9	28.72	midwest
4	Hallagan	2006-05-22	lip gloss	55	167.08	west
5	Zaret	2004-06-17	lip gloss	43	130.60	midwest

3.4.4. Grouping Data

Another useful method for inspecting data is to group it according to certain criteria. For the sales dataset, it would be useful to categorize all of the data by location, independent of the year. The `groupby` function in Pandas allows us to accomplish just that. This function returns a special grouped DataFrame as a result. As a result, an aggregate function must be used to create a suitable DataFrame. As a result, **all values in the same group will be subjected to this function**.

For instance, in our scenario, we can get a DataFrame that shows the number (count) of the transactions for each location across all years by grouping by location and using the count function as the aggregation technique for each group. As a consequence, a DataFrame with locations as indexes and counting values of transactions as the column would be created:

```
print(type(sales[['Location','Dollars']].groupby('Location')))

sales[['Location','Dollars']].groupby('Location').count()
```

```
<class 'pandas.core.groupby.generic.DataFrameGroupBy'>
```

Location	Dollars
east	456
midwest	424
south	521
west	490

```
# a DataFrame with locations as indexes and mean of sales income  as
# the column would be created

sales[['Location','Dollars']].groupby('Location').mean()
```

Location	Dollars
east	125.815965
midwest	129.258113
south	123.409616
west	129.467082

```
sales[['Location','Dollars','Units']].groupby('Location').mean()
```

Location	Dollars	Units
east	125.815965	41.267544
midwest	129.258113	42.417453
south	123.409616	40.466411
west	129.467082	42.491837

3.4.5. Rearranging Data

Until now, our indices were merely a numerical representation of rows with little meaning. We can change the way our data is organized by redistributing indexes and columns for better data processing, which usually results in greater performance. Using the `pivot_table` function, we may rearrange our data. We can define which columns will be the new indexes, values, and columns.

3.4.5.1. Simplest Pivot table

An `index` is required even for the smallest pivot table. Let us utilize the location as our index in this case. It uses the `'mean'` aggregation function on all available numerical columns by default.

```
sales.pivot_table(index='Location')
```

Location	Dollars	Units
east	125.815965	41.267544
midwest	129.258113	42.417453
south	123.409616	40.466411
west	129.467082	42.491837

To display multiple indexes, we can pass a list to index:

```
sales.pivot_table(index=['Location','Product'])
```

Location	Product	Dollars	Units
east	**eye liner**	132.521304	43.513043
	foundation	120.755248	39.584158
	lip gloss	119.001134	38.989691
	lipstick	136.096279	44.697674
	mascara	125.406000	41.120000
midwest	**eye liner**	127.295980	41.745098
	foundation	139.133333	45.712644
	lip gloss	130.765316	42.936709
	lipstick	136.465472	44.811321
	mascara	117.995340	38.669903
south	**eye liner**	121.481947	39.831858
	foundation	121.983852	39.977778
	lip gloss	121.492632	39.824561
	lipstick	115.709273	37.909091
	mascara	133.528462	43.846154
west	**eye liner**	136.215000	44.735849
	foundation	137.521553	45.194175
	lip gloss	115.428197	37.795082
	lipstick	132.699643	43.589286
	mascara	129.339223	42.446602

On the pivot table, the values to index are the keys to group by. To achieve a different visual representation, you can change the order of the values. For example, we can look at average values by grouping the region with the product category.

```
sales.pivot_table(index=['Product','Location'])
```

		Dollars	Units
Product	**Location**		
eye liner	**east**	132.521304	43.513043
	midwest	127.295980	41.745098
	south	121.481947	39.831858
	west	136.215000	44.735849
foundation	**east**	120.755248	39.584158
	midwest	139.133333	45.712644
	south	121.983852	39.977778
	west	137.521553	45.194175
lip gloss	**east**	119.001134	38.989691
	midwest	130.765316	42.936709
	south	121.492632	39.824561
	west	115.428197	37.795082
lipstick	**east**	136.096279	44.697674
	midwest	136.465472	44.811321
	south	115.709273	37.909091
	west	132.699643	43.589286
mascara	**east**	125.406000	41.120000
	midwest	117.995340	38.669903
	south	133.528462	43.846154
	west	129.339223	42.446602

3.4.5.2. Specifying values and performing aggregation

The mean aggregation function is applied to all numerical columns by default, and the result is returned. Use the `values` argument to specify the columns we are interested in.

```
sales.pivot_table(index=['Location'],
                        values = ['Dollars'])
```

	Dollars
Location	
east	125.815965
midwest	129.258113
south	123.409616
west	129.467082

We can specify a valid string function to `aggfunc` to perform an aggregation other than mean, for example, a sum:

```
sales.pivot_table(index=['Location'],
                        values = ['Dollars'], aggfunc = 'sum')
```

	Dollars
Location	
east	57372.08
midwest	54805.44
south	64296.41
west	63438.87

`aggfunc` can be a dict, and below is the dict equivalent.

```
sales.pivot_table(index=['Location'],
                        values = ['Dollars'], aggfunc = {'Dollars': 'sum'})
```

	Dollars
Location	
east	57372.08
midwest	54805.44
south	64296.41
west	63438.87

`aggfunc` can be a list of functions, and below is an example to display the `sum` and `count`

```
sales.pivot_table(index=['Location'],
                        values = ['Dollars'], aggfunc = ['sum','count'])
```

	sum	count
	Dollars	Dollars
Location		
east	57372.08	456
midwest	54805.44	424
south	64296.41	521
west	63438.87	490

3.4.5.3. Seeing break down using columns

If we would like to see sales broken down by product_category, the columns argument allows us to do that

```
sales.pivot_table(index=['Location'],
                  values = ['Dollars'],
                  aggfunc = 'sum',
                 columns='Product')
```

	Dollars				
Product	**eye liner**	**foundation**	**lip gloss**	**lipstick**	**mascara**
Location					
east	15239.95	12196.28	11543.11	5852.14	12540.60
midwest	12984.19	12104.60	10330.46	7232.67	12153.52
south	13727.46	16467.82	13850.16	6364.01	13886.96
west	14438.79	14164.72	14082.24	7431.18	13321.94

Note If there are missing values and we want to replace them, we could use `fill_value` argument, for example, to set `NaN` to a specific value.

```
sales.pivot_table(index=['Location'],
                  values = ['Dollars'],
                  aggfunc = 'sum',
                 columns='Product',
                 fill_value = 0)
```

	Dollars				
Product	**eye liner**	**foundation**	**lip gloss**	**lipstick**	**mascara**
Location					
east	15239.95	12196.28	11543.11	5852.14	12540.60
midwest	12984.19	12104.60	10330.46	7232.67	12153.52
south	13727.46	16467.82	13850.16	6364.01	13886.96
west	14438.79	14164.72	14082.24	7431.18	13321.94

3.4.6. Exercise

Apply `pivot_table` (in new worksheets) to answer the following questions.

1. The number of sales transactions for the given salesperson
2. For the given salesperson, the total revenue by the given product
3. Total revenue generated by the given salesperson broken down by the given location
4. Total revenue by the salesperson and the given year

3.4.7. Solutions to Exercise

1. The number of sales transactions for the given salesperson

```
sales.columns
```

```
Index(['Name', 'Date', 'Product', 'Units', 'Dollars', 'Location'], dtype='object')
```

```
sales.pivot_table(index=['Name'],
                  values=['Dollars'], aggfunc='count')
```

	Dollars
Name	
Ashley	197
Betsy	217
Cici	230
Colleen	206
Cristina	207
Emilee	203
Hallagan	200
Jen	217
Zaret	214

```
sales.pivot_table(index=['Name'],
                  values=['Dollars'], aggfunc = 'count').sort_values(by = 'Dollars')
```

	Dollars
Name	
Ashley	197
Hallagan	200
Emilee	203
Colleen	206
Cristina	207
Zaret	214
Betsy	217
Jen	217
Cici	230

1. For the given salesperson, the total revenue by the given product

```
sales.pivot_table(index='Name',
                  values='Dollars',
                   columns = 'Product', aggfunc = 'sum',  margins=True)
```

Product	eye liner	foundation	lip gloss	lipstick	mascara	All
Name						
Ashley	5844.95	4186.06	6053.67	3245.46	6617.10	25947.24
Betsy	6046.51	8043.48	5675.65	3968.62	4827.22	28561.48
Cici	5982.83	6198.22	5199.96	3148.83	7060.71	27590.55
Colleen	3389.61	6834.76	5573.35	2346.39	6746.54	24890.65
Cristina	5397.30	5290.97	5298.00	2401.68	5461.64	23849.59
Emilee	7587.37	5313.82	5270.26	2189.15	4719.34	25079.94
Hallagan	6964.64	6985.80	5603.10	3177.88	5703.32	28434.74
Jen	7010.45	5628.65	5461.65	3953.28	6887.21	28941.24
Zaret	8166.73	6451.66	5670.33	2448.71	3879.94	26617.37
All	56390.39	54933.42	49805.97	26880.00	51903.02	239912.80

The result above also show the total. In Panda pivot_table(), we can simply pass `margins=True`.

1. Total revenue generated by the given salesperson broken down by the given location

```
sales.pivot_table(index=['Name','Location'],
                  values='Dollars', aggfunc = 'sum').head(10)
```

		Dollars
Name	**Location**	
Ashley	**east**	7772.70
	midwest	4985.90
	south	7398.58
	west	5790.06
Betsy	**east**	8767.41
	midwest	4878.07
	south	7732.04
	west	7183.96
Cici	**east**	5956.31
	midwest	8129.60

1. Total revenue by the salesperson and the given year.

To generate a yearly report with Panda pivot_table(), here are the steps:

1. Defines a groupby instruction using Grouper() with key='Date' and freq='Y'.
2. Applies pivot_table.

Note To group our data depending on the specified frequency for the specified column, we'll use `pd.Grouper(key=INPUT COLUMN>, freq=DESIRED FREQUENCY>)`. The frequency in our situation is 'Y' and the relevant column is 'Date.'

Different standard frequencies, such as 'D','W','M', or 'Q', can be used instead of 'Y.' Check out the following for a list of less common useable frequencies, https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#offset-aliases.

```
year_gp = pd.Grouper(key='Date',freq='Y')
print(year_gp)

sales.pivot_table(index='Name', columns=year_gp, values='Dollars',aggfunc='sum')
```

```
TimeGrouper(key='Date', freq=<YearEnd: month=12>, axis=0, sort=True, closed='right',
label='right', how='mean', convention='e', origin='start_day')
```

Date	2004-12-31	2005-12-31	2006-12-31
Name			
Ashley	9495.09	9547.53	6904.62
Betsy	9420.24	9788.70	9352.54
Cici	8965.25	9024.96	9600.34
Colleen	9361.37	7996.81	7532.47
Cristina	9132.12	7976.34	6741.13
Emilee	7805.66	9326.44	7947.84
Hallagan	10676.88	9102.52	8655.34
Jen	9049.31	8920.30	10971.63
Zaret	9078.51	8639.68	8899.18

Exercise

Apply pivot_table (in new worksheets) to answer the following questions.

1. How many saleperson are there? Hint: use `groupby` to create a grouped DataFrame grouped by salepersons and then call `groups` on the grouped object, which will returns the list of indices for every group.

```
grouped_person = sales.groupby('Name')
```

```
print(grouped_person.groups.keys())
len(grouped_person.groups.keys())
```

```
dict_keys(['Ashley', 'Betsy', 'Cici', 'Colleen', 'Cristina', 'Emilee', 'Hallagan', 'Jen',
'Zaret'])
```

```
9
```

```
grouped_person.size()
```

```
Name
Ashley      197
Betsy       217
Cici        230
Colleen     206
Cristina    207
Emilee      203
Hallagan    200
Jen         217
Zaret       214
dtype: int64
```

Previous
2. Python Basics

Next
4. Data Visualization

By Pairote Satiracoo