PennState World Campus

DAAN862: Analytics Programming in Python

Lesson 3: Statistical Analysis with Pandas

Lesson 3: Statistical Analysis with Pandas

Pandas contain high-level data structures and manipulation tools designed to make data analysis fast and easy in Python. Pandas are built on top of NumPy which makes it easy to use in NumPy-centric applications.

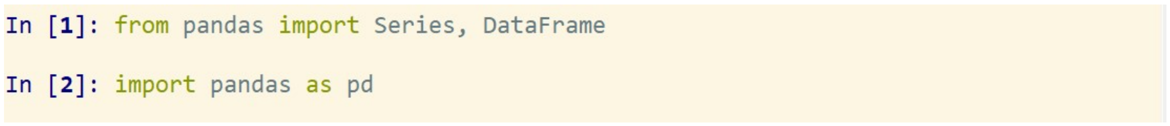
At the end of this lesson, students will be able to:

- Create two basic data types in pandas: Series and DataFrame
- Utilize methods of Series and DataFrame in pandas
- Utilize pandas dataset import and export functions
- Apply statistic analysis in pandas to Series and DataFrame with pandas

By the end of this lesson, please complete all readings and assignments found in the Lesson 3 Course Schedu le.

Lesson 3.1: Basic Data Types: Series

In this course, the import conventions shown in Figure 3.1 are used for pandas.



```
In [1]: from pandas import Series, DataFrame

In [2]: import pandas as pd
```

Fig 3.1 (click to enlarge)

pd is a common short name for pandas. You can also import *Series* and *DataFrame* into local name space such that you can directly use function *Series* or *DataFrame* instead of *pd.Series* or *pd.DataFrame* which are only used for convenience, however both ways are acceptable.

A ***Series*** is a one-dimensional array-like object containing an array of data (of any NumPy data type) and has an associated array of data labels, called its ***index***. The simplest Series is formed from only an array of data:

Fig 3.2

The string representation of a series displayed in Fig 3.2 interactively shows the index on the left and the values on the right. Since we did not specify an index of data, a default one consisting of the integers 0 through N-1 (where N is the length of the data) is created.

You can customize this index by assigning a list of strings to the index parameter in Series function:

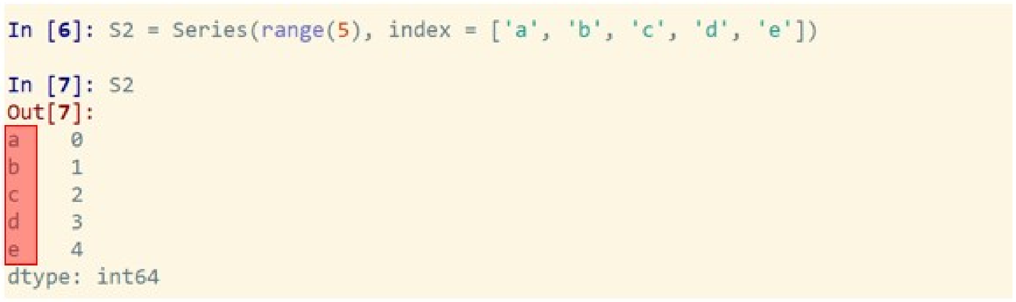

```
In [6]: S2 = Series(range(5), index = ['a', 'b', 'c', 'd', 'e'])

In [7]: S2
Out[7]:
a    0
b    1
c    2
d    3
e    4
dtype: int64
```

Fig 3.3 (click to enlarge)

As highlighted in Figure 3.3, the indexes are ['a','b','c','d','e'].

As showing in Figure 3.4, series can be indexed similarly to list or array. It can also be indexed by the values in the index:

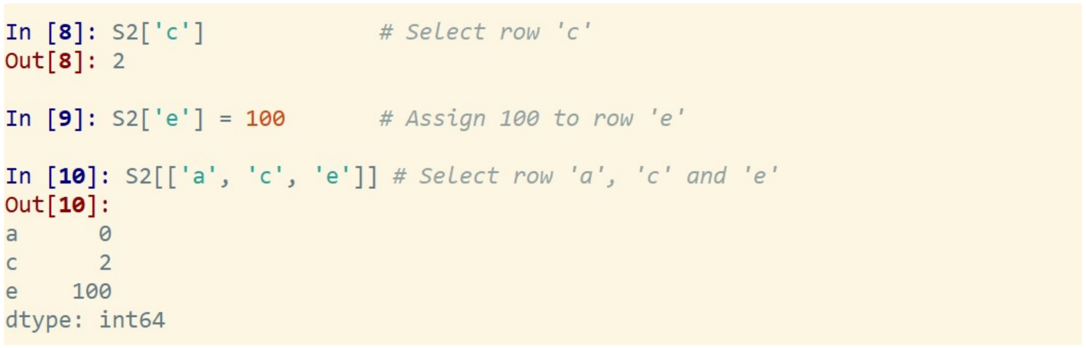
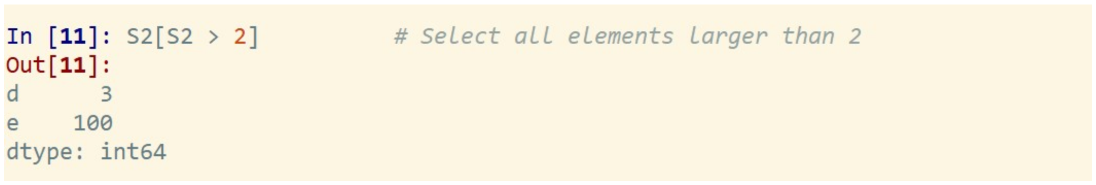

```
In [8]: S2['c']             # Select row 'c'
Out[8]: 2

In [9]: S2['e'] = 100       # Assign 100 to row 'e'

In [10]: S2[['a', 'c', 'e']] # Select row 'a', 'c' and 'e'
Out[10]:
a      0
c      2
e    100
dtype: int64
```

Fig 3.4 (click to enlarge)

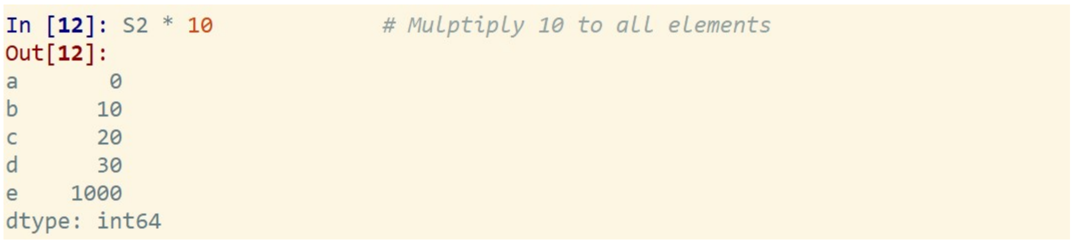
As shown in Figure 3.5, similar logic comparison and math operations can be applied to Series:



```
In [11]: S2[S2 > 2]          # Select all elements larger than 2
Out[11]:
d      3
e    100
dtype: int64
```

Fig 3.5 (click to enlarge)

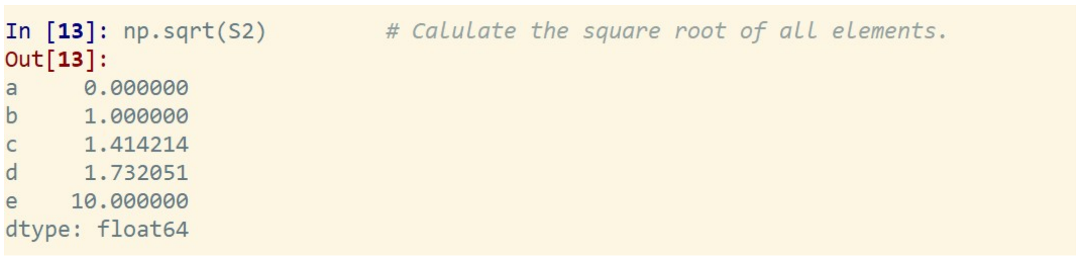
As showing in Figure 3.6, you can multiply 10 to all elements:



```
In [12]: S2 * 10             # Mulptiply 10 to all elements
Out[12]:
a       0
b      10
c      20
d      30
e    1000
dtype: int64
```

Fig 3.6 (click to enlarge)

As shown in Figure 3.7, the NumPy functions can be applied to Series too. For example, you can calculate the square root of all elements:



```
In [13]: np.sqrt(S2)         # Calulate the square root of all elements.
Out[13]:
a     0.000000
b     1.000000
c     1.414214
d     1.732051
e    10.000000
dtype: float64
```

Fig 3.7 (click to enlarge)

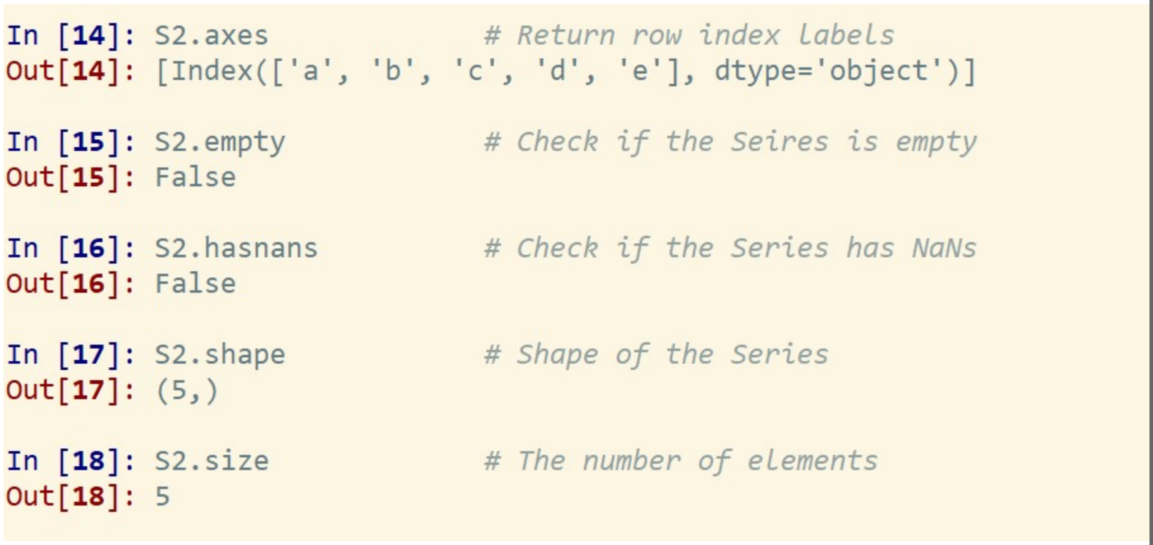
Table 3.1.1: The Important Attributes of Series

Attributes	**Description**
T	Return the transpose, which is by definition self
at	Fast label-based scalar accessor
axes	**Return a list of the row axis labels**
dtype	Return the dtype object of the underlying data
empty	**Return Boolean if the series is empty**
hasnans	**Return Boolean if the series contain nans**
iat	Fast integer location scalar accessor

Attributes	Description
iloc	Purely integer-location based indexing for selection by position
is unique	Return boolean if values in the object are unique
loc	Purely label-location based indexer for selection by label
ndim	Return the number of dimensions of the underlying data
shape	**Return a tuple of the shape of the underlying data**
size	**Return the number of elements in the underlying data**
values	Return Series as ndarray or ndarray-like

Example

Click the dropdown to view examples on how to use some of the attributes shown in Table 3.1.1



```
In [14]: S2.axes             # Return row index labels
Out[14]: [Index(['a', 'b', 'c', 'd', 'e'], dtype='object')]

In [15]: S2.empty            # Check if the Seires is empty
Out[15]: False

In [16]: S2.hasnans          # Check if the Series has NaNs
Out[16]: False

In [17]: S2.shape            # Shape of the Series
Out[17]: (5,)

In [18]: S2.size             # The number of elements
Out[18]: 5
```

Fig 3.8 (click to enlarge)

Lesson 3.2: Basic Data Types: Data Frame (3 of 7)

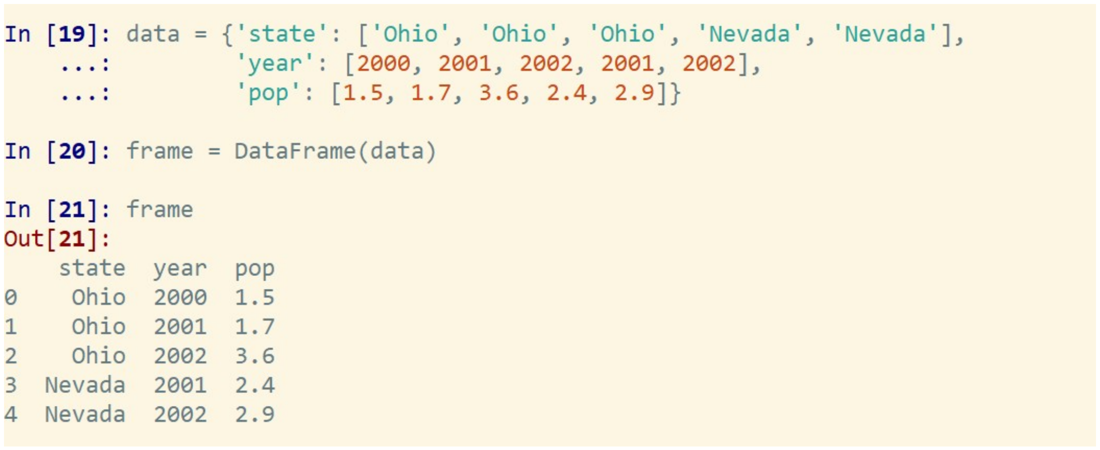
Lesson 3.2: Basic Data Types: Data Frame

Another basic data type in Python is data frame, which is equivalent to the dat frames in R. A ***DataFrame*** is a table of data which use columns and rows to represent variables and instances, respectively. The DataFrame has both a column and row index.

(click the tabs to learn more)

Creating a DatatFrame with dict

In Figure 3.9, data is dict which has three keys ['state', 'year', and 'pop'] and three lists as their values. When you create a DataFrame from this dict, it will use dict keys as column names and dict values as elements.

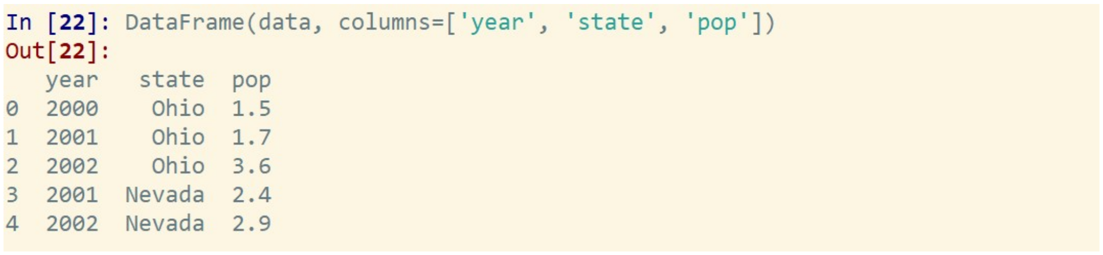


```
In [19]: data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
    ...:         'year': [2000, 2001, 2002, 2001, 2002],
    ...:         'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}

In [20]: frame = DataFrame(data)

In [21]: frame
Out[21]:
    state  year  pop
0    Ohio  2000  1.5
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
```

Fig 3.9 (click to enlarge)

Specifying Sequence of Columns

As shown in Figure 3.10, by default it will arrange the column using alphabetic order. You can specify the sequence of columns by using the column arguments:

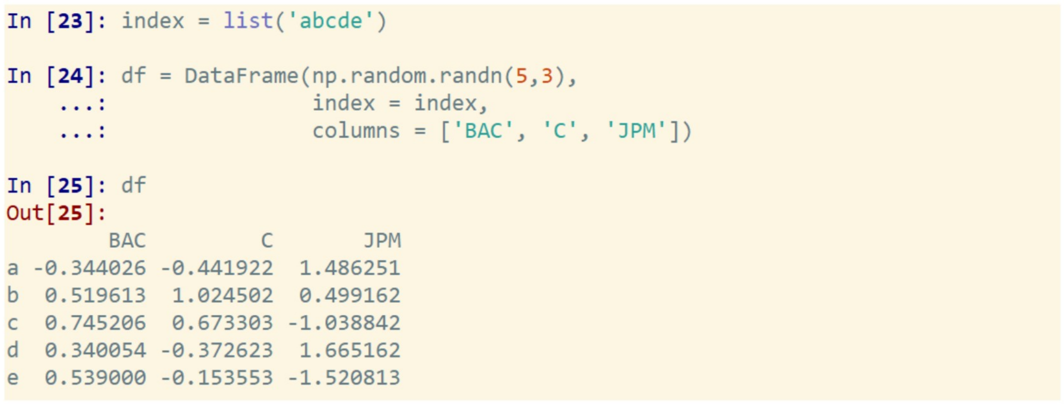


```
In [22]: DataFrame(data, columns=['year', 'state', 'pop'])
Out[22]:
   year   state  pop
0  2000    Ohio  1.5
1  2001    Ohio  1.7
2  2002    Ohio  3.6
3  2001  Nevada  2.4
4  2002  Nevada  2.9
```

Fig 3.10 (click to enlarge)

Creating a DataFrame with Numpy Array

It is easy to convert a Numpy array to a DataFrame. The created DataFrame will have the same dimension as the Numpy array and you can specify the index and columns by yourself. Figure 3.11 shows an example of converting a random Numpy array with the size of (5, 3) to a DataFrame. You can also specify the index and column labels, as shown in Figure 3.11:



```
In [23]: index = list('abcde')

In [24]: df = DataFrame(np.random.randn(5,3),
    ...:                index = index,
    ...:                columns = ['BAC', 'C', 'JPM'])

In [25]: df
Out[25]:
        BAC         C       JPM
a -0.344026 -0.441922  1.486251
b  0.519613  1.024502  0.499162
c  0.745206  0.673303 -1.038842
d  0.340054 -0.372623  1.665162
e  0.539000 -0.153553 -1.520813
```

Fig 3.11 (click to enlarge)

DataFrame has similar attributes as series which will be introduced in the next sections.

Transcript

In this video, I will walk through two basic data types of pandas.

The first one is Series, which is a one-dimension array-like object.

Let’s import pandas first. You can import pandas as pd. This is a conventional way or you can choose a different name other than pd.

For convenience, we can just import Series, and DataFrame from pandas. In this case, you just need to type Series and DataFrame instead of pd.Series or pd.DataFrame.

You can convert a vector like data to series. Here, we convert range(5) to series. The left side is its indexes and the right side is its values.

By default, it will use consecutive integers starts from 0 as the index. But you can also specify a different index by add the argument index = something. Here we use character a, b, c, d,e as its index.

If your index is consecutive integers, you can select elements of Series similar to list. Otherwise, you can put the index labels into square brackets to select elements. Like what we show here.

S2[‘c’] will select row ‘c’

S2[‘e’] will select row ‘e’

You can also put an array of index labels here to select multiple elements.

Logical and math operation is similar to numpy array, which use vectorized or elementwise operations. You can also apply numpy functions to Series. Np.sqrt(S2) will apply square root to all elements of S2.

There are some common methods of Series， here we show a few example.

S2.axes will return row index labels.

S2.empty will check if S2 is empty or not.

S2.hasnans will check if s2 has nans

S2.shape return the shape of Series

S2.size return the size of Series. Shape method return a tuple and size method return a number.

The next data type of pandas is DataFrame, which is a similar data structure like the data.frame in R. typically, each column represents a variable and each row represents an instance.

You can create a dataFrame by convert a dictionary. Each key-value pair will become column name and column values. Here we have a dictionary called data. Which has three keys. The values for each keys is a list. Now, you can convert this dictionary to DataFrame by using DataFrame function. We can see that state, year and pop become three columns.

You can specify the column order by using columns arguments and pass the correct order to it.

Here, we change the order to year, state, pop.

You can also convert numpy array to a dataframe.

First, we create a vector called index

This np.random.randn(5, 3) will create a 5x3 random array. And we specify its index as abcde and use column names called ‘BAC’, ‘C’, ‘JPM’.

We can see how df looks like: It use the random array as the values and index vector as index labels and this as column names.

You can get the column names by using columns attribute and get the shapes by using df.shape.

Lesson 3.3: Indexing and Slicing

In this section, we will introduce basic operations of Series and DataFrame: Reindexing, Dropping Data, Indexing and Selection.

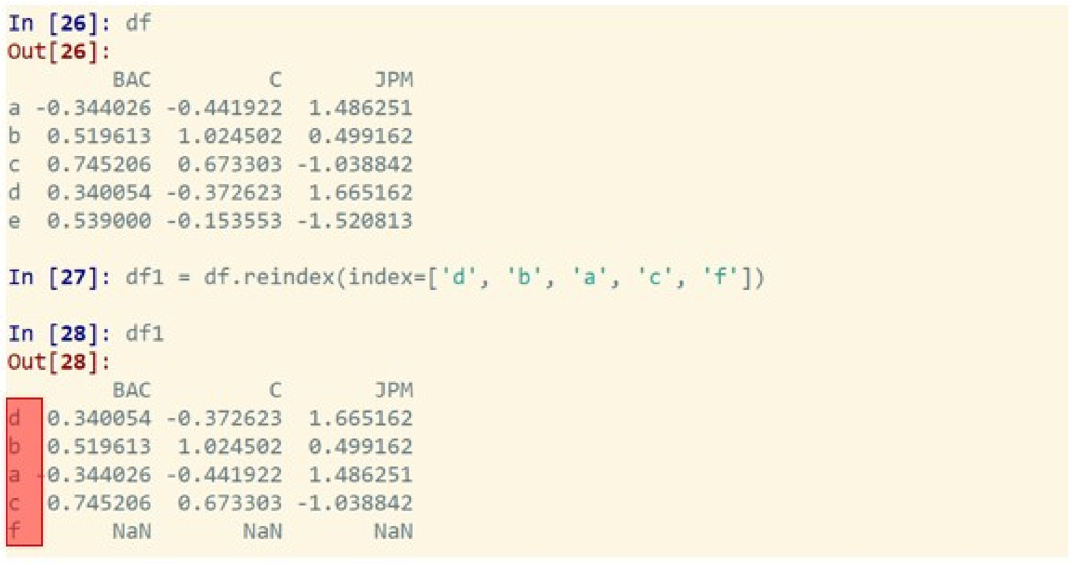
(click the titles to learn more)

Reindexing

Reindex is a critical method in pandas. By using this method, a new object will be created with a new data. If any value of a new index is not present in the data then missing values will be introduced.

Examples

Reindexing a DataFrame



```
In [26]: df
Out[26]:
        BAC         C       JPM
a -0.344026 -0.441922  1.486251
b  0.519613  1.024502  0.499162
c  0.745206  0.673303 -1.038842
d  0.340054 -0.372623  1.665162
e  0.539000 -0.153553 -1.520813

In [27]: df1 = df.reindex(index=['d', 'b', 'a', 'c', 'f'])

In [28]: df1
Out[28]:
        BAC         C       JPM
d  0.340054 -0.372623  1.665162
b  0.519613  1.024502  0.499162
a -0.344026 -0.441922  1.486251
c  0.745206  0.673303 -1.038842
f       NaN       NaN       NaN
```

Fig 3.12 (click to enlarge)

As highlighted in Figure 3.12, the index has been reset to ['d','b','a','c','f'] and since 'f' isn't presented in the original data, NaNs has been introduced.

Reset index

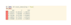


Fig 3.13

You can reset the index used integer-based indexes by using ***reset index***

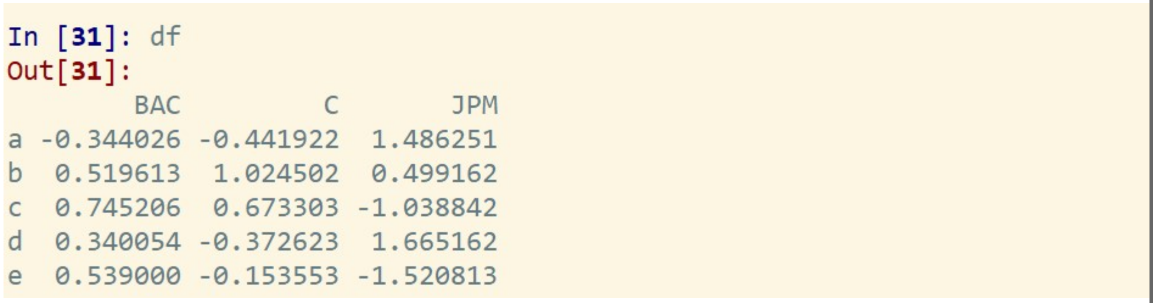
As highlighted in Figure 3.13, the index becomes 0,1, ..., 4. drop= True argument will drop the original index. If using drop= False (by default), the original index will be appended in a new column.

Dropping Entries From An Axis

Dropping one or more rows or columns is easy by using ***drop method***. The drop method will return a new object with the indicated value or values deleted from an axis. Let's continue using the df created in Figure 3.14:

Examples

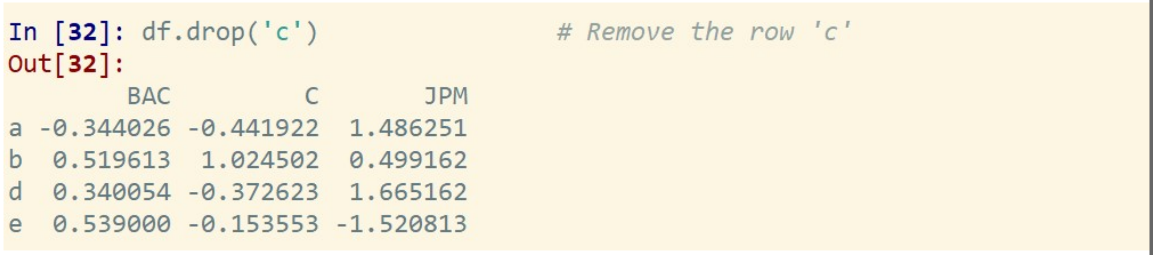
Dropping a single row by its index:



```
In [31]: df
Out[31]:
        BAC         C       JPM
a -0.344026 -0.441922  1.486251
b  0.519613  1.024502  0.499162
c  0.745206  0.673303 -1.038842
d  0.340054 -0.372623  1.665162
e  0.539000 -0.153553 -1.520813
```

Fig 3.14 (click to enlarge)

Dropping multiple rows

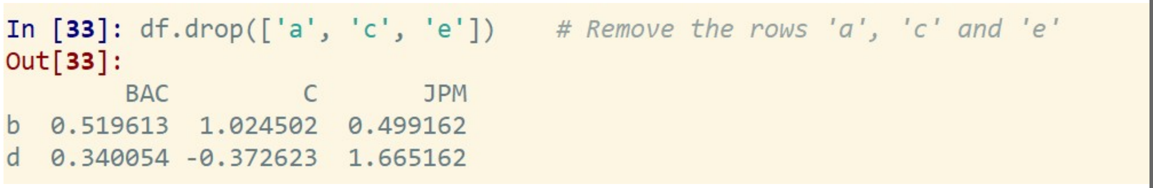


```
In [32]: df.drop('c')              # Remove the row 'c'
Out[32]:
        BAC         C       JPM
a -0.344026 -0.441922  1.486251
b  0.519613  1.024502  0.499162
d  0.340054 -0.372623  1.665162
e  0.539000 -0.153553 -1.520813
```

Fig 3.15 (click to enlarge)

You can remove row 'c' as shown in Figure 3.15 by using the method *drop.*

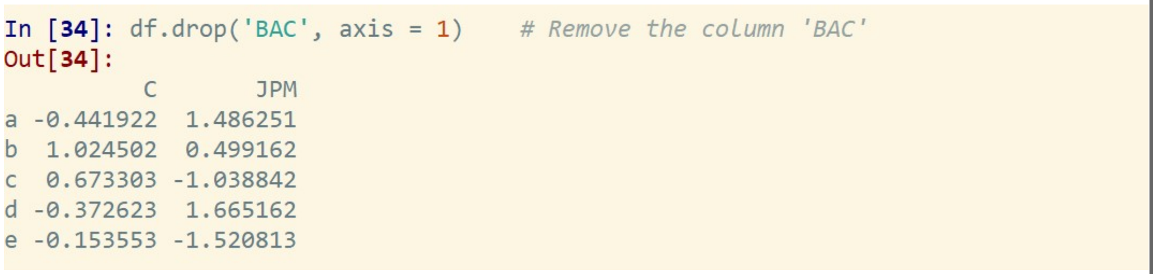
You can also drop multiple rows ['a','c','e']:



```
In [33]: df.drop(['a', 'c', 'e'])    # Remove the rows 'a', 'c' and 'e'
Out[33]:
        BAC         C       JPM
b  0.519613  1.024502  0.499162
d  0.340054 -0.372623  1.665162
```

Fig 3.16 (click to enlarge)

Dropping a column instead of a row



```
In [34]: df.drop('BAC', axis = 1)    # Remove the column 'BAC'
Out[34]:
          C       JPM
a -0.441922  1.486251
b  1.024502  0.499162
c  0.673303 -1.038842
d -0.372623  1.665162
e -0.153553 -1.520813
```

Fig 3.17 (click to enlarge)

By setting *axis=1*, you will drop a column instead of a row

Indexing & Selection

Indexing Series and DataFrame works similarly to NumPy array indexing except that you can also use the index values instead of only integers.

There are several ways to select a column or several columns:

DataFrame[*variable name]*: Since the variable name is the label of columns, you can directly use this format

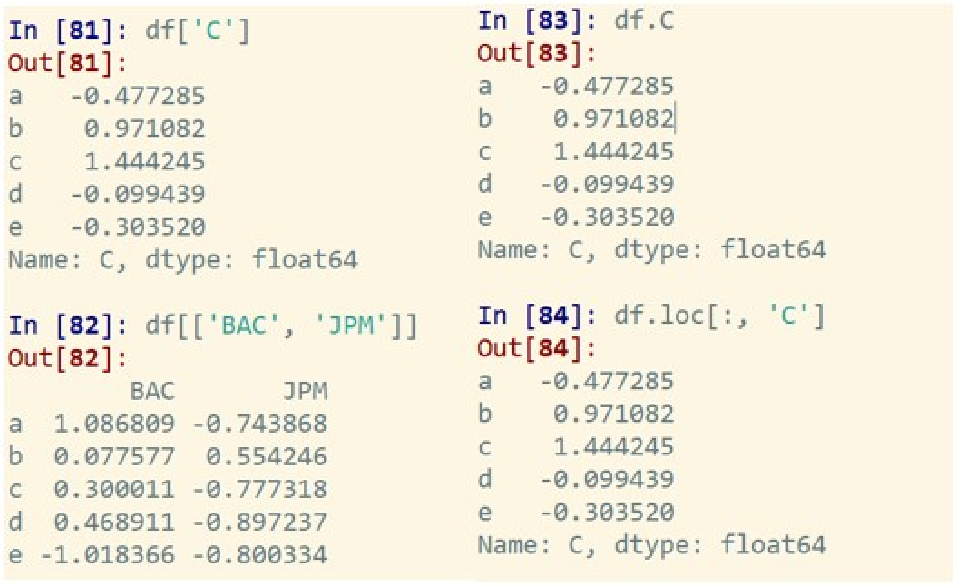
DataFrame.[*variable.name*]: Since variable becomes the attribute of the DataFrame, you can use "." and followed by the variable name.

DataFrame.loc[row labels, column labels] This method is used to locate the element with row label and column label. If row or column labels parameter is missing, all row or columns will be selected.

DataFrame.iloc[*row index, column index*]: This method is used to locate elements with row index and column index (integer-based index). If row or column index paramet is missing, all row or columns will be selected.

Examples

Figure 3.18 shows an example of using these ways to select column "C":

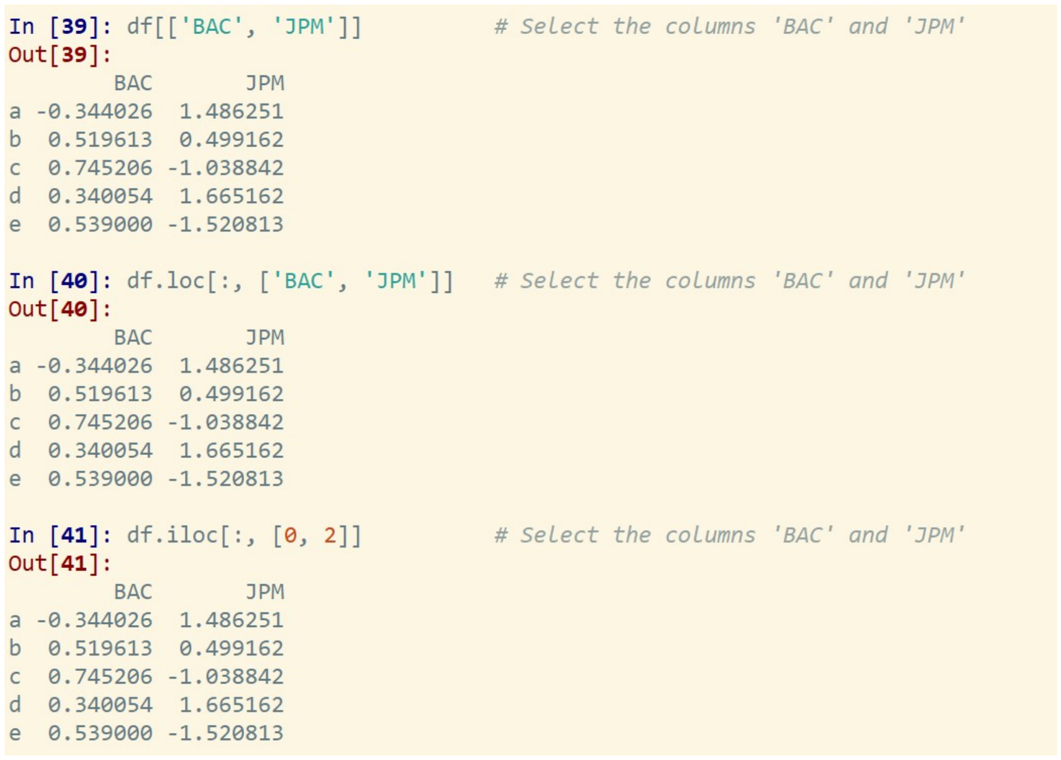


```
In [81]: df['C']
Out[81]:
a   -0.477285
b    0.971082
c    1.444245
d   -0.099439
e   -0.303520
Name: C, dtype: float64

In [82]: df[['BAC', 'JPM']]
Out[82]:
        BAC       JPM
a  1.086809 -0.743868
b  0.077577  0.554246
c  0.300011 -0.777318
d  0.468911 -0.897237
e -1.018366 -0.800334
```

```
In [83]: df.C
Out[83]:
a   -0.477285
b    0.971082
c    1.444245
d   -0.099439
e   -0.303520
Name: C, dtype: float64

In [84]: df.loc[:, 'C']
Out[84]:
a   -0.477285
b    0.971082
c    1.444245
d   -0.099439
e   -0.303520
Name: C, dtype: float64
```

Fig 3.18 (click to enlarge)

Similarly, you can select multiple rows as shown in Figure 3.19:



```
In [39]: df[['BAC', 'JPM']]          # Select the columns 'BAC' and 'JPM'
Out[39]:
        BAC       JPM
a -0.344026  1.486251
b  0.519613  0.499162
c  0.745206 -1.038842
d  0.340054  1.665162
e  0.539000 -1.520813

In [40]: df.loc[:, ['BAC', 'JPM']]   # Select the columns 'BAC' and 'JPM'
Out[40]:
        BAC       JPM
a -0.344026  1.486251
b  0.519613  0.499162
c  0.745206 -1.038842
d  0.340054  1.665162
e  0.539000 -1.520813

In [41]: df.iloc[:, [0, 2]]          # Select the columns 'BAC' and 'JPM'
Out[41]:
        BAC       JPM
a -0.344026  1.486251
b  0.519613  0.499162
c  0.745206 -1.038842
d  0.340054  1.665162
e  0.539000 -1.520813
```

Fig 3.19 (click to enlarge)

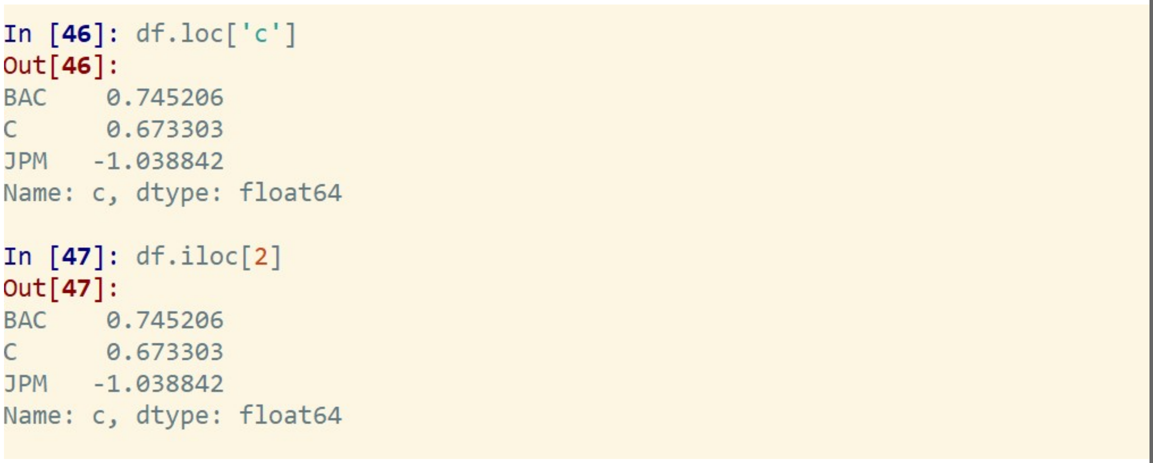
In Figure 3.19, there are only two ways to select rows: *loc* and *iloc.* These ways were mentioned at the beginning of Indexing & Selection (this section).

Click dropdown to view

DataFrame.loc[row labels, column labels] This method is used to locate the element with row label and column label. If row or column labels parameter is missing, all row or columns will be selected.

DataFrame.iloc[*row index, column index*]: This method is used to locate elements with row index and column index (integer-based index). If row or column index paramet is missing, all row or columns will be selected.

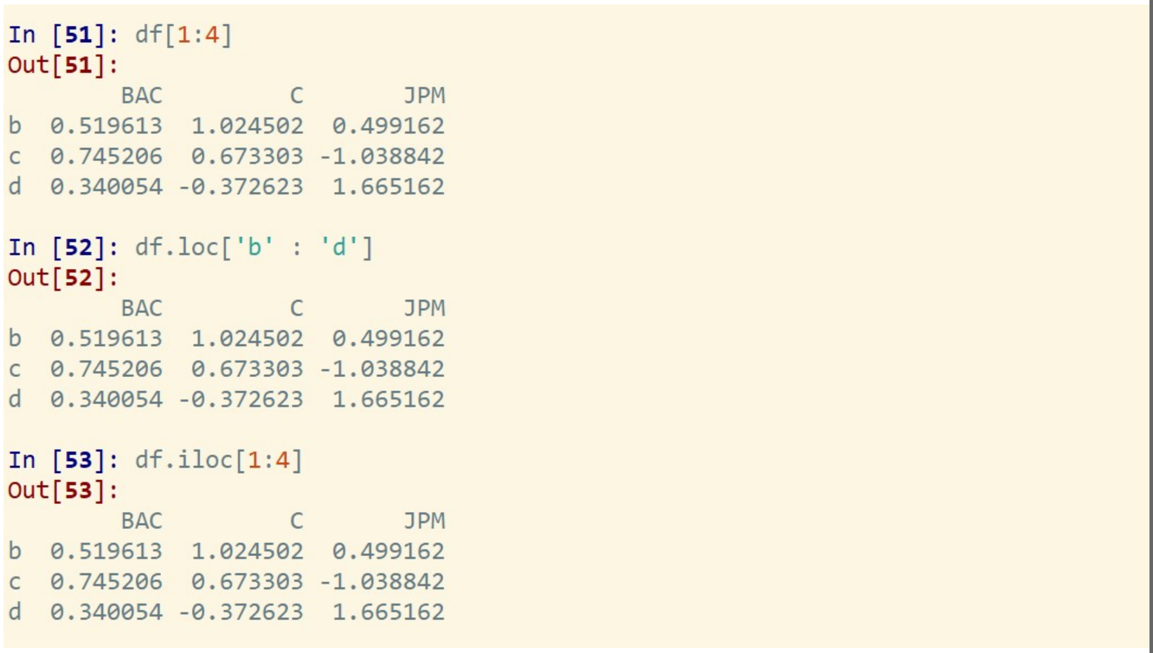
Figure 3.20 is an example of how to select the 2nd row:



```
In [46]: df.loc['c']
Out[46]:
BAC    0.745206
C      0.673303
JPM   -1.038842
Name: c, dtype: float64

In [47]: df.iloc[2]
Out[47]:
BAC    0.745206
C      0.673303
JPM   -1.038842
Name: c, dtype: float64
```

Fig 3.20 (click to enlarge)

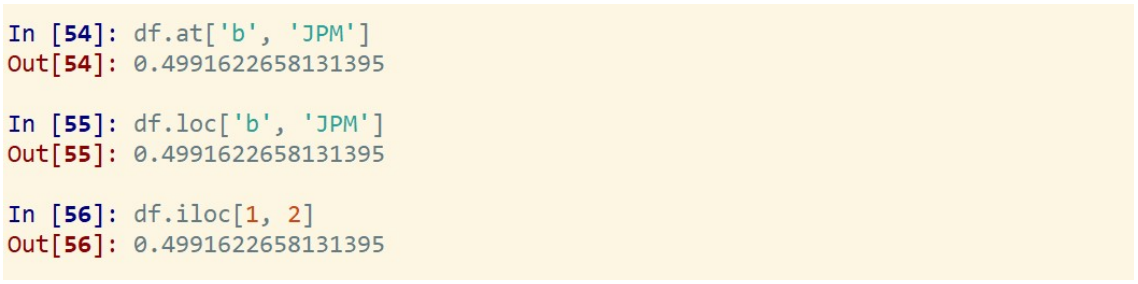
You can also select multiple rows ['b', 'c', 'd'] as shown in Figure 3.21:



```
In [51]: df[1:4]
Out[51]:
        BAC         C       JPM
b  0.519613  1.024502  0.499162
c  0.745206  0.673303 -1.038842
d  0.340054 -0.372623  1.665162

In [52]: df.loc['b' : 'd']
Out[52]:
        BAC         C       JPM
b  0.519613  1.024502  0.499162
c  0.745206  0.673303 -1.038842
d  0.340054 -0.372623  1.665162

In [53]: df.iloc[1:4]
Out[53]:
        BAC         C       JPM
b  0.519613  1.024502  0.499162
c  0.745206  0.673303 -1.038842
d  0.340054 -0.372623  1.665162
```

Fig 3.21 (click to enlarge)

You can also get the value of an element. Figure 3.22 shows an example of getting the value from row 'b' and column 'JPM':



```
In [54]: df.at['b', 'JPM']
Out[54]: 0.4991622658131395

In [55]: df.loc['b', 'JPM']
Out[55]: 0.4991622658131395

In [56]: df.iloc[1, 2]
Out[56]: 0.4991622658131395
```

Fig 3.22 (click to enlarge)

Transcript

In this video, I am going to introduce how to index and slice the dataFrame

The first method called reindex. You can change the index label by using reindex method. You can pass the new index label to index argument.

Df1 is the result after we reindex the df with d,c, a, c f. and The data will be rearranged according to this order. If an index label doesn't appear in the original data, na will be filled for the row with new labels, like row f here.

If you want to reset the index with consecutive integers starting with 0, you need to use reset_index function. After passing drop= True , the original index will be dropped, otherwise, a new column will created for the original index labels.

Next we show how to slice the data Frame.

If you would like to remove rows or columns, you should use drop method. You can list row labels in the method to remove one or multiple rows. Line 73 is used to remove row c. For multiple rows, you can put the list of row labels here. Line 74 removes row ace. If you would like to remove columns, you need to add axis = 1 in the method. By default axis = 0, which means remove row. Here we can remove column BAC by running line 75.

There are several ways to select columns.

1, you can use square brackets and put the column name inside of them, like line 79 show here.

2. You can also use dot plus the column name since columns become attributes of DataFrame.

3. You can also use loc function, you should put row labels before the comma and put column names after the comma. If you use a colon, which means you will select all rows or all columns.

4. You can use iloc function which is similar to loc function, the only difference is that you put row or column number inside of the function.

You can select multiple columns by using a list instead of one element here. Line 85- 87 are used to select multiple columns.

If you want to select one row, you can only use loc or iloc function. The comma and colon can be omitted.

Line 93 selects row c

Line 94 selects the third row.

For multiple consecutive rows, you can use start label or number colon end label or number.

If you would select column and row at the same time, you can use loc or iloc and fill the row and column information in these methods.

Line 102 select row bc on column “C’

You can use at, loc, iloc to select one element.

DataFrame also supports Boolean selection. If you would like to select rows, you can put Boolean array before the comma, for columns you can put Boolean array after the comma.

Line 113 creates a Boolean array of if elements in column C are large than 0 or not.

You can use to select rows, as shown in line 114.

Line 115 creates a Boolean of if elements in row c are larger than 0 or not.

You can use this to select column as shown in line 116

Lesson 3.4: Import and Export Data (5 of 7)

Lesson 3.4: Import and Export Data

Pandas features a number of functions for reading tabular data as a DataFrame object. The pandas I/O API is a set of top-level reader functions accessed liked *pd.read csv*() that generally return a pandas object. The corresponding writer functions are object methods that are accessed like *df.to csv*().

Import Data:

For import functions, we only introduce common ones: read csv, read table, and read excel.

The details of read csv function are listed below. You can specify the delimer, header, names, index col, etc in the function.

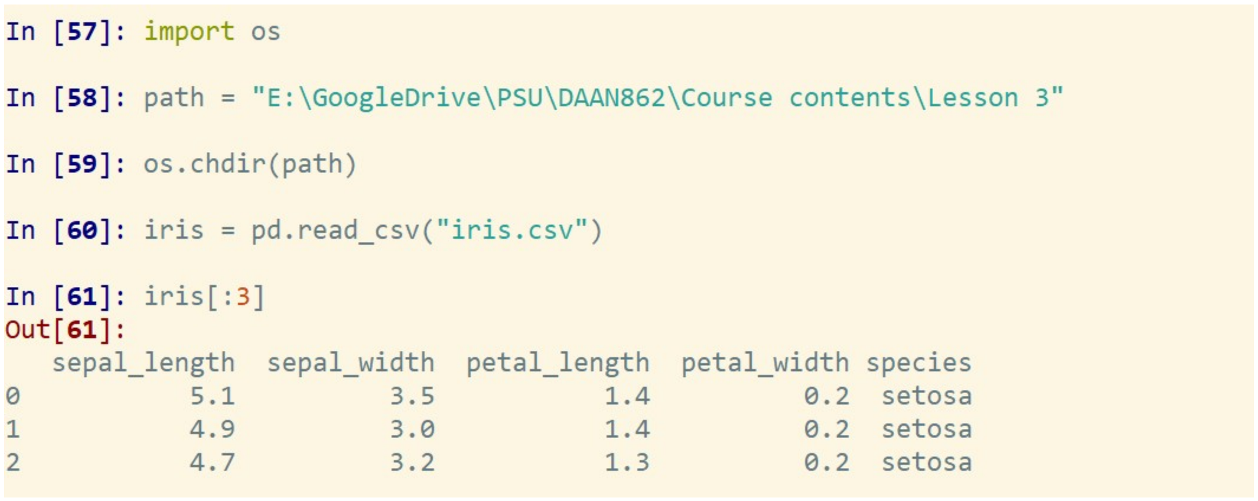
Click the dropdown to view details of read csv

pandas.read_csv(*filepath_or_buffer*, *sep=', '*, *delimiter=None*, *header='infer'*, *names=None*, *index_col=None*, *usecols=None*, *squeeze=False*, *prefix=None*, *mangle_dupe_cols=True*, *dtype=None*, *engine=None*, *converters=None*, *true_values=None*, *false_values=None*, *skipinitialspace=False*, *skiprows=None*, *nrows=None*, *na_values=None*, *keep_default_na=True*, *na_filter=True*, *verbose=False*, *skip_blank_lines=True*, *parse_dates=False*, *infer_datetime_format=False*, *keep_date_col=False*, *date_parser=None*, *dayfirst=False*, *iterator=False*, *chunksize=None*, *compression='infer'*, *thousands=None*, *decimal=b'.'*, *lineterminator=None*, *quotechar='"'*, *quoting=0*, *escapechar=None*, *comment=None*, *encoding=None*, *dialect=None*, *tupleize_cols=None*, *error_bad_lines=True*, *warn_bad_lines=True*, *skipfooter=0*, *skip_footer=0*, *doublequote=True*, *delim_whitespace=False*, *as_recarray=None*, *compact_ints=None*, *use_unsigned=None*, *low_memory=True*, *buffer_lines=None*, *memory_map=False*, *float_precision=None*)

(click the tabs to learn more)

Read CSV:

Figure 3.23 shows an example of importing iris.csv with read csv:



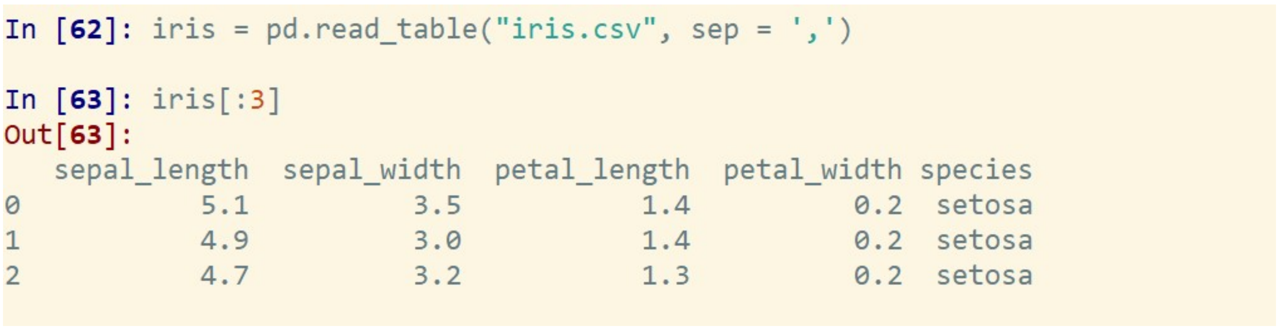
```
In [57]: import os

In [58]: path = "E:\GoogleDrive\PSU\DAAN862\Course contents\Lesson 3"

In [59]: os.chdir(path)

In [60]: iris = pd.read_csv("iris.csv")

In [61]: iris[:3]
Out[61]:
   sepal_length  sepal_width  petal_length  petal_width species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
2           4.7          3.2           1.3          0.2  setosa
```

Fig 3.23 (click to enlarge)

Since the data is a standard csv file just use the default settings.

Read Table:

Figure 3.24 shows an example of using a read table and specifying the delimiter:

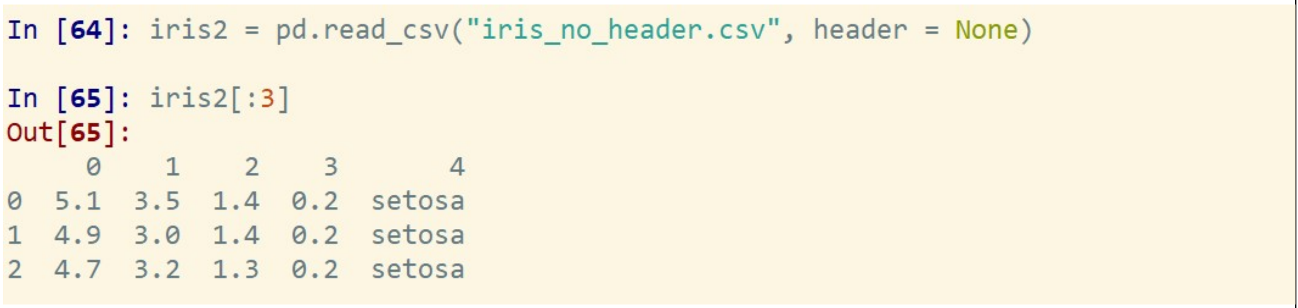


```
In [62]: iris = pd.read_table("iris.csv", sep = ',')

In [63]: iris[:3]
Out[63]:
   sepal_length  sepal_width  petal_length  petal_width species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
2           4.7          3.2           1.3          0.2  setosa
```

Fig 3.24 (click to enlarge)

No Header:

Figure 3.25 shows an example of data with no header:

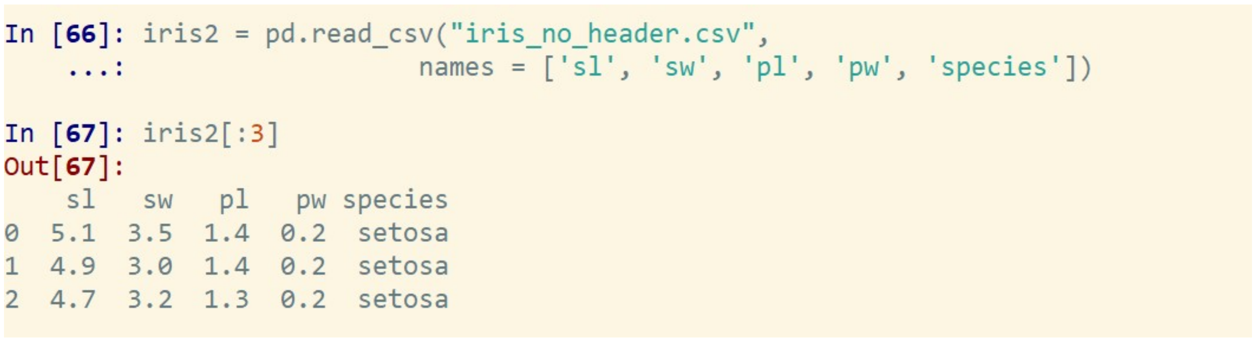


```
In [64]: iris2 = pd.read_csv("iris_no_header.csv", header = None)

In [65]: iris2[:3]
Out[65]:
     0    1    2    3       4
0  5.1  3.5  1.4  0.2  setosa
1  4.9  3.0  1.4  0.2  setosa
2  4.7  3.2  1.3  0.2  setosa
```

Fig 3.25 (click to enlarge)

Specify Headers:

Figure 3.26 shows an example of how you can specify headers by yourself:

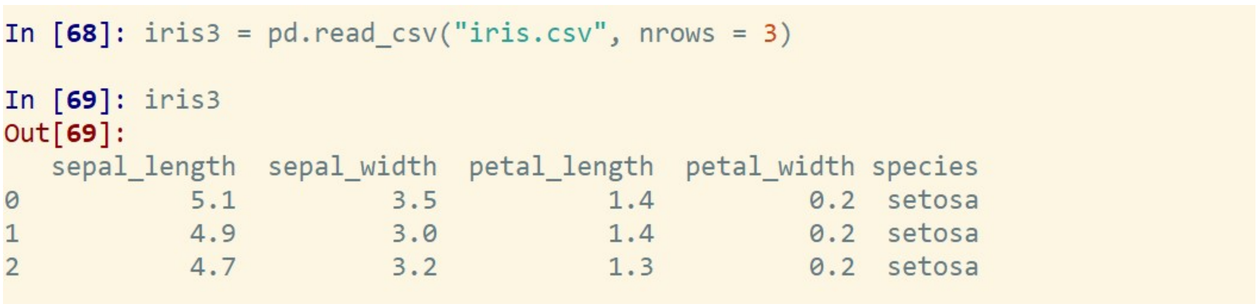


```
In [66]: iris2 = pd.read_csv("iris_no_header.csv",
    ...:                     names = ['sl', 'sw', 'pl', 'pw', 'species'])

In [67]: iris2[:3]
Out[67]:
    sl   sw   pl   pw species
0  5.1  3.5  1.4  0.2  setosa
1  4.9  3.0  1.4  0.2  setosa
2  4.7  3.2  1.3  0.2  setosa
```

Fig 3.26 (click to enlarge)

N-Rows:

If you want to only read out a small number of rows (avoiding the entire file) you are able to specify that with nrows as shown in Figure 3.27:

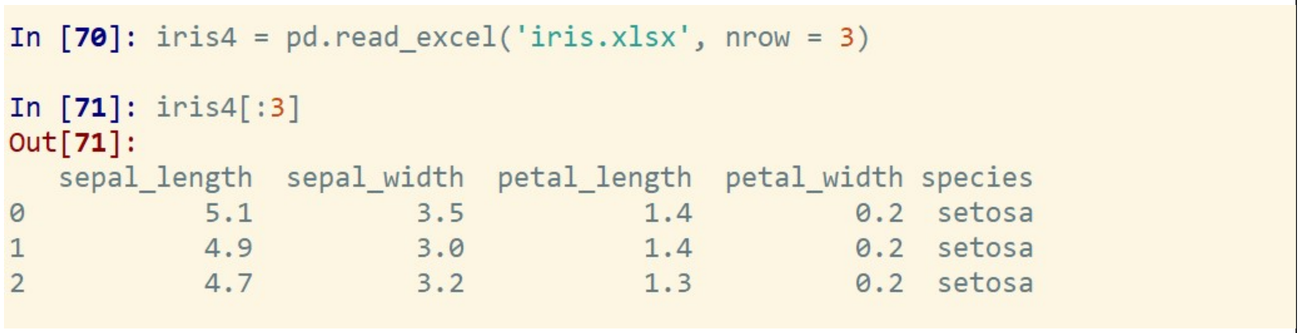


```
In [68]: iris3 = pd.read_csv("iris.csv", nrows = 3)

In [69]: iris3
Out[69]:
   sepal_length  sepal_width  petal_length  petal_width species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
2           4.7          3.2           1.3          0.2  setosa
```

Fig 3.27 (click to enlarge)

Read Excel:

For importing Excel files, you can use *read_excel* function:



```
In [70]: iris4 = pd.read_excel('iris.xlsx', nrow = 3)

In [71]: iris4[:3]
Out[71]:
   sepal_length  sepal_width  petal_length  petal_width species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
2           4.7          3.2           1.3          0.2  setosa
```

Fig 3.28 (click to enlarge)

Export Data:

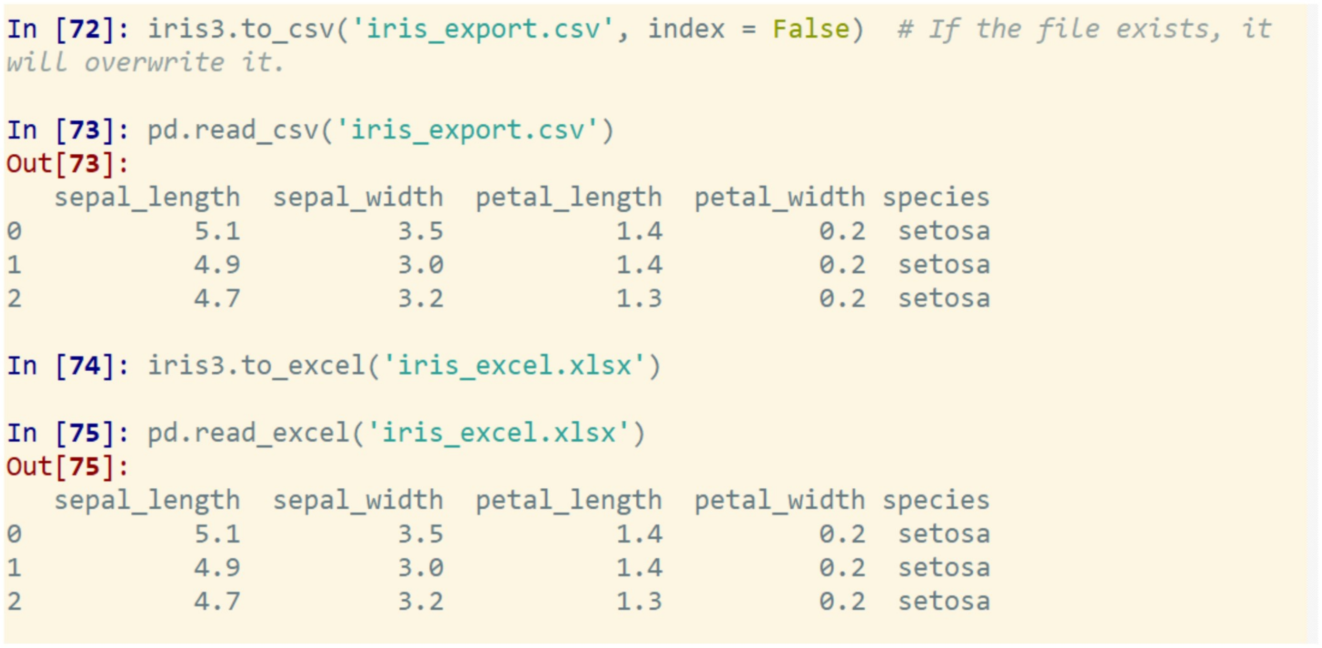
pandas provides various data format for exporting, such as csv, xlsx, sql, json etc. In this lesson, we will only focus on csv and xlsx format.

Click the dropdown to view details of DataFrame.to_csv and DataFrame.to_excel

DataFrame.**to_csv**(*path_or_buf=None*, *sep=', '*, *na_rep=''*, *float_format=None*, *columns=None*, *header=True*, *index=True*, *index_label=None*, *mode='w'*, *encoding=None*, *compression=None*, *quoting=None*, *quotechar='"'*, *line_terminator='\n'*, *chunksize=None*, *tupleize_cols=None*, *date_format=None*, *doublequote=True*, *escapechar=None*, *decimal='.'*)

DataFrame.**to_excel**(*excel_writer*, *sheet_name='Sheet1'*, *na_rep=''*, *float_format=None*, *columns=None*, *header=True*, *index=True*, *index_label=None*, *startrow=0*, *startcol=0*, *engine=None*, *merge_cells=True*, *encoding=None*, *inf_rep='inf'*, *verbose=True*, *freeze_panes=None*)

Figure 3.29 shows examples of the CSV and Excel files:



```
In [72]: iris3.to_csv('iris_export.csv', index = False)  # If the file exists, it
will overwrite it.

In [73]: pd.read_csv('iris_export.csv')
Out[73]:
   sepal_length  sepal_width  petal_length  petal_width species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
2           4.7          3.2           1.3          0.2  setosa

In [74]: iris3.to_excel('iris_excel.xlsx')

In [75]: pd.read_excel('iris_excel.xlsx')
Out[75]:
   sepal_length  sepal_width  petal_length  petal_width species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
2           4.7          3.2           1.3          0.2  setosa
```

Fig 3.29 (click to enlarge)

Transcript

In this video, I will show how to import and export csv and excel files.

First, you need to import os package. And define the path as the folder directory of where you saved the file. Keep in mind you need to use forwardslash instead of backslash in the directory.

os.chdir() is a function to change the directory to the input.

Read_csv is function to import csv files into Python. We can take a look at the first three rows.

If your file is not a standard csv file. Like separated by space or something else. You can specify the separations by using sep argument.

If the data is no header, you can add argument header = None in the read_csv function.

You can also customer the column names by using argument names = a list of character.

Here we use sl, sw, pl, pw species as the column names for iris data.

If you would like to import partial data, you can use argument nrow, here we only import 3 lines.

There are other arguments for read_csv function, you can choose to use them based on your data.

Read_excel is the function to import excel data into Python. It has similar argument as the read_csv function. Here we only try nrow = 3.

If you would like export data into csv or excel, you can use the built-in methods for DataFrame: to_csv or to_excel, you need to give the file name and the file will be saved in the current working directory.

Lesson 3.5: Statistical Analysis

Pandas objects contains common mathematical and statistical methods. By specifying the axis, you can perform summary statistics (such as mean, sum, std, etc.) for the rows or columns. Compared to NumPy arrays, they are all built to exclude missing data.

See Table 3.5.1 for the full list of summary statistics and related methods.

Table 3.5.1: List of Summary Statistics and Related Methods

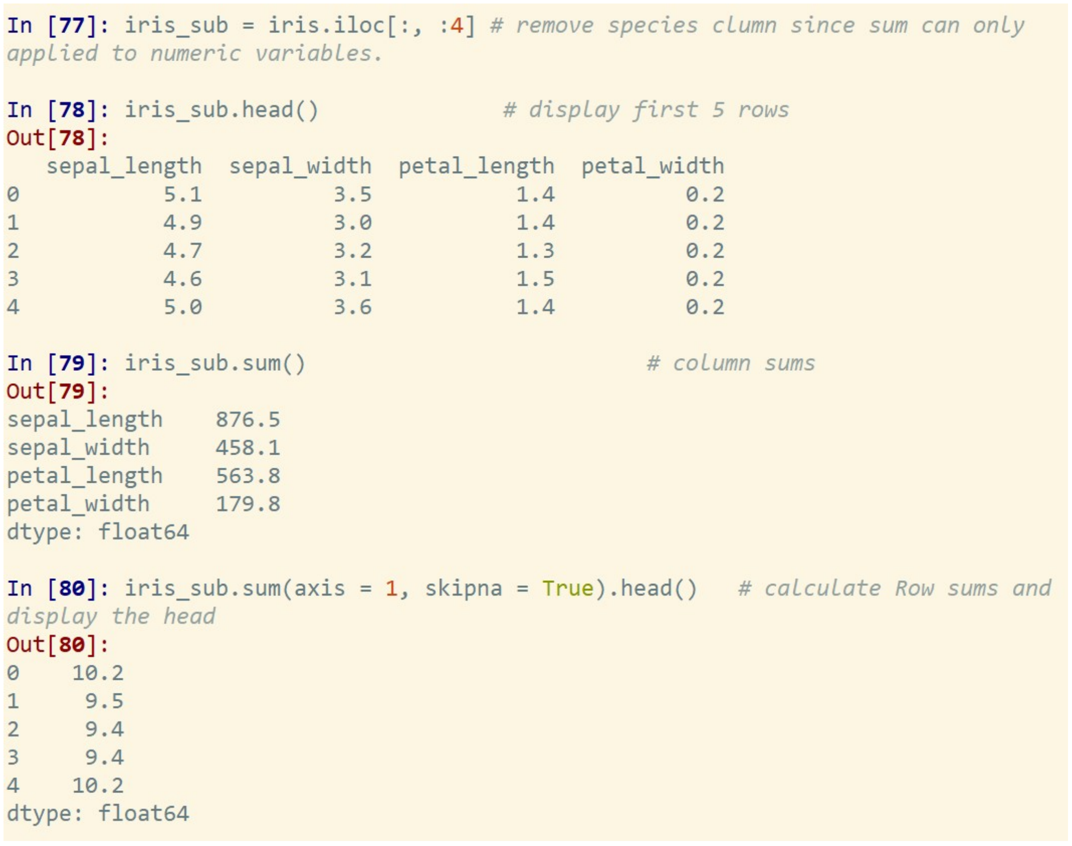
Method	Description
count	Number of non-NA values
describe	Compute set of summary statistics for Series or each DataFrame column
min, max	Compute minimum and maximum values

Method	Description
idxmin, idxmax	Compute index values at which minimum or maximum value obtained, respectively
quantile	Compute sample quantile ranging from 0 to 1
sum	Sum of values
mean	Mean of values
median	Arithmetic median (50% quantile) of values
mad	Mean absolute deviation from mean value
var	Sample variance of values
std	Sample standard deviation of values
skew	Sample skewness (3rd moment) of values
kurt	Sample kurtosis (4th moment) of values
cumsum	Cumulative sum of values
cummin, cummax	Cumulative minimum or maximum of values, respectively
cumprod	Cumulative product of values
diff	Compute 1st arithmetic difference (useful for time series)
pct change	Compute percent changes

Examples

Click the dropdown to view example of sum method

Figure 3.30 shows an example of how to use a sum method which returns a Series with column sums.



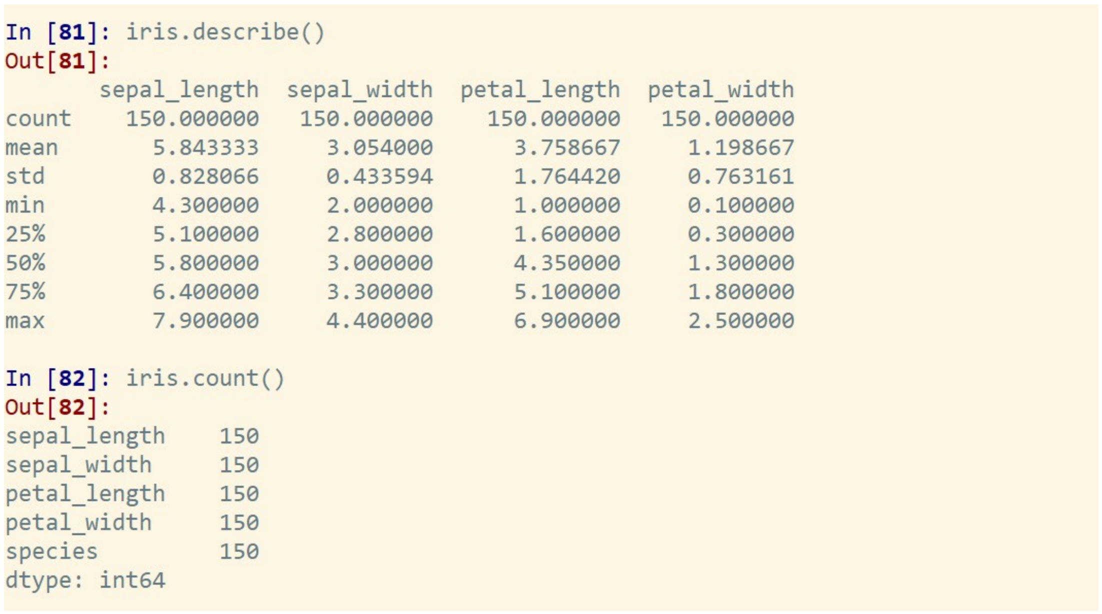
```
In [77]: iris_sub = iris.iloc[:, :4] # remove species clumn since sum can only
applied to numeric variables.

In [78]: iris_sub.head()              # display first 5 rows
Out[78]:
   sepal_length  sepal_width  petal_length  petal_width
0           5.1          3.5           1.4          0.2
1           4.9          3.0           1.4          0.2
2           4.7          3.2           1.3          0.2
3           4.6          3.1           1.5          0.2
4           5.0          3.6           1.4          0.2

In [79]: iris_sub.sum()                           # column sums
Out[79]:
sepal_length    876.5
sepal_width     458.1
petal_length    563.8
petal_width     179.8
dtype: float64

In [80]: iris_sub.sum(axis = 1, skipna = True).head()   # calculate Row sums and
display the head
Out[80]:
0    10.2
1     9.5
2     9.4
3     9.4
4    10.2
dtype: float64
```

Fig 3.30 (click to enlarge)

Passing axis=1 calculates the sums over the rows instead

NA values can be excluded by using the skipna option

Click the dropdown for a summary of a DataFrame column



```
In [81]: iris.describe()
Out[81]:
       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

In [82]: iris.count()
Out[82]:
sepal_length    150
sepal_width     150
petal_length    150
petal_width     150
species         150
dtype: int64
```

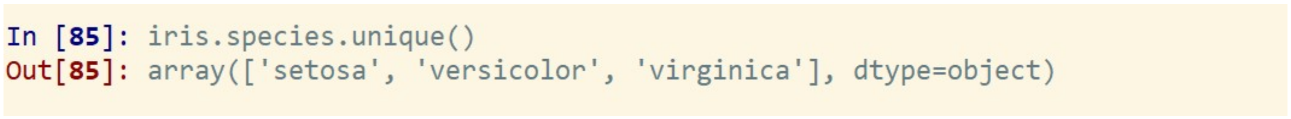
Fig 3.31 (click to enlarge)

Figure 3.31 shows the describe method, which computes a set of summary for each DataFrame column.

(click the tabs to learn more)

Unique Values, Value Counts, and Membership

Another class of statistics related methods gives information about one-dimensional Series as shown in Figure 3.32. The first function is unique, which gives the unique values in a Series:

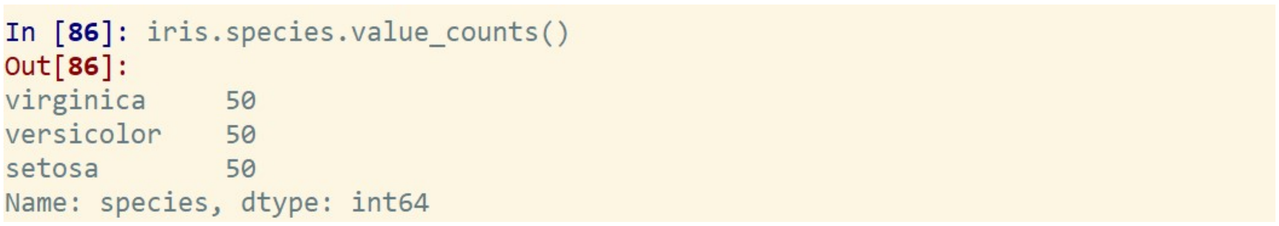


```
In [85]: iris.species.unique()
Out[85]: array(['setosa', 'versicolor', 'virginica'], dtype=object)
```

Fig 3.32 (click to enlarge)

Typically, the returned unique values are not sorted, but you can sort it afterwards by *uniques.sort*().

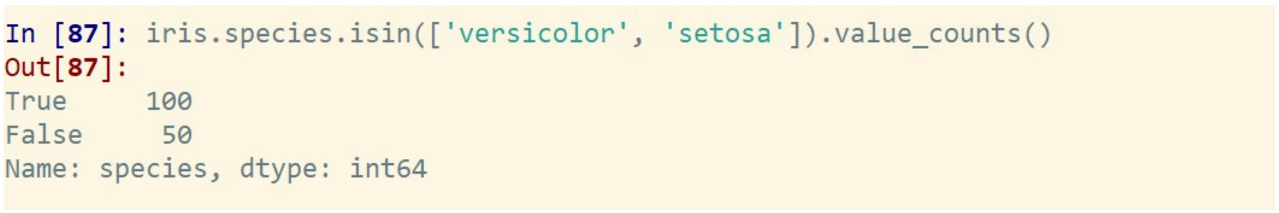
Relatedly, *value counts* returns a Series with value frequencies as shown in Figure 3.33:



```
In [86]: iris.species.value_counts()
Out[86]:
virginica     50
versicolor    50
setosa        50
Name: species, dtype: int64
```

Fig 3.33 (click to enlarge)

Isin performs a vectorized set membership check and can be used in filtering a dataset down to a subset of values in a Series or column in a DataFrame as shown in Figure 3.34:



```
In [87]: iris.species.isin(['versicolor', 'setosa']).value_counts()
Out[87]:
True     100
False     50
Name: species, dtype: int64
```

Fig 3.34 (click to enlarge)

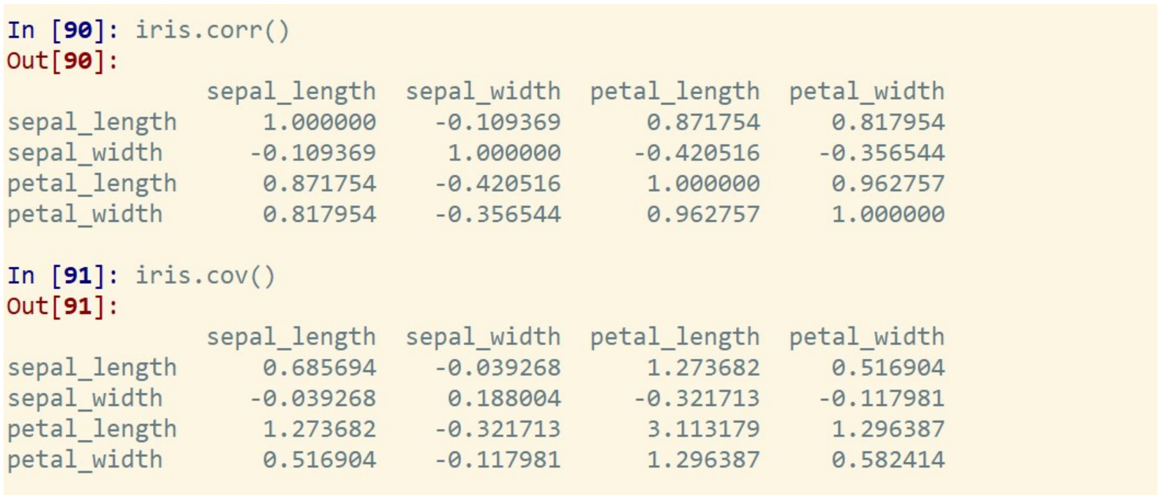
Correlation and Covariance

Correlation and Covariance are computed from pairs of arguments.

Figure 3.35 shows examples of how to calculate correlation and covariance of two columns:

Fig 3.35
(click to
enlarge)

For a full correlation or covariance matrix, you can use DataFrame's *corr* and *cov* methods instead as shown in Figure 3.36:



```
In [90]: iris.corr()
Out[90]:
              sepal_length  sepal_width  petal_length  petal_width
sepal_length      1.000000    -0.109369      0.871754     0.817954
sepal_width      -0.109369     1.000000     -0.420516    -0.356544
petal_length      0.871754    -0.420516      1.000000     0.962757
petal_width       0.817954    -0.356544      0.962757     1.000000

In [91]: iris.cov()
Out[91]:
              sepal_length  sepal_width  petal_length  petal_width
sepal_length      0.685694    -0.039268      1.273682     0.516904
sepal_width      -0.039268     0.188004     -0.321713    -0.117981
petal_length      1.273682    -0.321713      3.113179     1.296387
petal_width       0.516904    -0.117981      1.296387     0.582414
```

Fig 3.36 (click to enlarge)

Corrwith computes pairwise correlations between a DataFrame with another Series or DataFrame as shown in Figure 3.37:

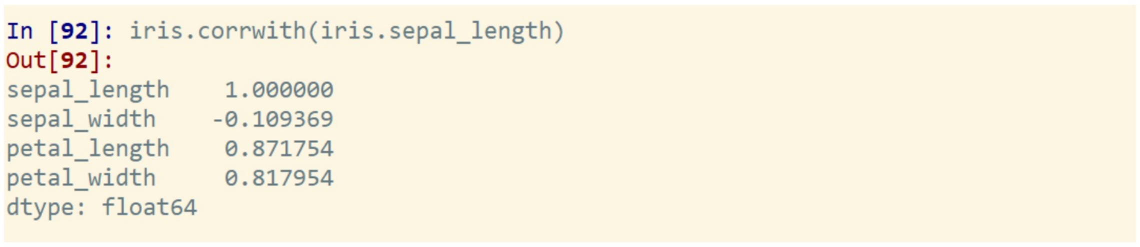

```
In [92]: iris.corrwith(iris.sepal_length)
Out[92]:
sepal_length    1.000000
sepal_width    -0.109369
petal_length    0.871754
petal_width     0.817954
dtype: float64
```

Fig 3.37 (click to enlarge)

Transcript

In this video, I am going to introduce the statisticanalysis with pandas.

First, let’s select numeric columns in iris data, called iris_sub. You can view the head of iris_sub data

DataFrame has built-in statistic method similar like Numpy array. Here, I only show sum function. You can also add argument axis = 1 for row sum and add skipna argument.

Describe shows the statistic information such as count, mean, std, min, minx, first qantile, median, third quantile and maximum at the same time.

Count method is to count the data points. From the result, we can see than it has 150 data points for each column. If there are NAs in the data, it can be reflected from count results.

Diff function is used to calcuate the difference between current row and the previous row. If we check iris_diff we can see that the first row are NANs since there is no previous row for it.

For categorical data, you can explore it use methods such as unique, value_counts.

Unique method returns unique values in the column. After running line 161, we can see that iris species column has three unique values: virginica, versicolor, and setosa.

Value_counts returns unique values and their counts at the same time. From the result of line 162, we can see that it has 50 data for each species.

You can also perform value_Counts to the boolean column.

Isin method is to check element of Species column in the list of versicolor and setosa and then or not and then perform a value count. The results show that we have 100 data in the list of versicolor and setosa and 50 data are not in.

Next we show how to calculate the correlation and covariace.

If you want to calculate the correlation and covariance between two columns, you can select the column first, then use its corr or cov method.

Line 166 calculates the correlation of sepal_length and sepal_width

Line 167 calculates the covariance of spepal_length and sepal_width

For correlation and covariance matrix, you can use the method for DataFrame, Like line 168 and line 169 shown here.

If you would like to calcuate the correlation between a column to a DataFrame. You can use dataFrame.corrwith and put the selected column inside the method. Like the line 170 shown here.

Lesson 3 References (7 of 7)

Lesson 3 References

- https://pandas.pydata.org/pandas-docs/stable/tutorials.html (https://pandas.pydata.org/pandas-docs/stable/tutorials.html)
- https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.html (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.html)
- “Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython”, Author Wes McKinney, Publisher "O'Reilly Media, Inc.", 2012