CS112
Introduction to Python Programming

Session 11: Pandas

Shengwei Hou
Ph.D., Assistant Professor
Department of Ocean Science and Engineering
Fall 2022

SUSTech Southern University of Science and Technology

明德求是 日新自强
VIRTUE | TRUTH | ADVANCE

Contents

SUSTech Southern University of Science and Technology

- Reading from csv files
- Reading from text files
- Reading from excel files
- Date and time in pandas
- Series
- Time Series
- DataFrame Introduction
- DataFrame Creation
- Extracting information
- Grouping and Aggregation
- Iterating over groups

Introductions

- `Pandas` is a powerful Python package for manipulating and analyzing time series and large & small data sets
- `Pandas` is designed to make working with "relational" or "labeled" data both easy and intuitive
- It has a spreadsheet-like character and aims to be the fundamental high-level building block for doing practical, real world data analysis
- The two primary data structures of pandas, Series (one-dimensional) and DataFrame (two-dimensional), handle the vast majority of typical use cases in many areas of science and engineering
- Routines available in `Pandas` can be accessed by:

```
>>> import pandas as pd
```

Reading from csv files

- `Pandas` can read data from files written in many different formats such as text, csv, Excel, JSON (JavaScript Object Notation), fixed-width text tables, HTML (web pages)
- `Pandas` function `pd.read_csv()` reads the data into a special `Pandas` object called a DataFrame
- A `DataFrame` is a tabular data structure similar to a spreadsheet. It is the central data structure of `Pandas`

ScatMieData.csv

```
Wavelength [vacuum] (nm) = 532,,
Refractive index of solvent = 1.33,,
Refractive index of particles = 1.59,,
Diameter of particles (microns) = 0.5,,
Cos_theta,F1,F2
1.00E+00,7.00E+01,7.00E+01
8.75E-01,2.71E+01,2.35E+01
7.50E-01,8.58E+00,6.80E+00
6.25E-01,1.87E+00,1.72E+00
5.00E-01,2.25E-01,5.21E-01
3.75E-01,3.04E-01,3.11E-01
2.50E-01,6.54E-01,2.36E-01
1.25E-01,7.98E-01,1.49E-01
0.00E+00,7.04E-01,7.63E-02
```

Reading from csv files

SUSTech Southern University of Science and Technology

```
>>> import pandas as pd
>>> scat = pd.read_csv('ScatMieData.csv', skiprows=4)
>>> type(scat)
<class 'pandas.core.frame.DataFrame'>
>>> scat
>>> scat.Cos_theta
>>> scat['Cos_theta']
>>> scat.Cos_theta[2]
0.75
>>> scat.Cos_theta[2:5]
2    0.750
3    0.625
4    0.500
Name: Cos_theta, dtype: float64
>>> scat['Cos_theta'][2]
0.75
```

columns

index

	Cos_theta	F1	F2
0	1.000	70.0000	70.0000
1	0.875	27.1000	23.5000
2	0.750	8.5800	6.8000
3	0.625	1.8700	1.7200
4	0.500	0.2250	0.5210
5	0.375	0.3040	0.3110
6	0.250	0.6540	0.2360
7	0.125	0.7980	0.1490
8	0.000	0.7040	0.0763
9	-0.125	0.4850	0.0406
10	-0.250	0.2650	0.0364

Reading from csv files


```
>>> head = pd.read_csv('ScatMieData.csv', nrows=4, header=None)
>>> head
                                        0   1   2
0         Wavelength [vacuum] (nm) = 532 NaN NaN
1     Refractive index of solvent = 1.33 NaN NaN
2   Refractive index of particles = 1.59 NaN NaN
3  Diameter of particles (microns) = 0.5 NaN NaN
>>> head[0][1]
'Refractive index of solvent = 1.33'
>>> head[1][1]
nan
```

Reading from csv files

- With `read_table()`, the user can specify the symbol that will be used to separate the columns of data, i.e., a symbol other than a comma can be used

```
>>> head = pd.read_table('ScatMieData.csv', sep='=', nrows=4, comment=',', header=None)
>>> head
                                      0       1
0          Wavelength [vacuum] (nm)  532.00
1       Refractive index of solvent    1.33
2     Refractive index of particles    1.59
3  Diameter of particles (microns)     0.50
>>> head[0][0]
'Wavelength [vacuum] (nm) '
>>> head[1][:]
0    532.00
1      1.33
2      1.59
3      0.50
```

`comment=',' :` Everything following "," will be taken as the comment part.

Reading from csv files

- If you prefer for the columns to be labeled by descriptive names instead of numbers, you can use the keyword names to provide names for the columns:

```
>>> head = pd.read_table('ScatMieData.csv', sep='=', nrows=4,
comment=',', header=None, names=['property', 'value'])
>>> head
                              property   value
0          Wavelength [vacuum] (nm)   532.00
1       Refractive index of solvent     1.33
2     Refractive index of particles     1.59
3  Diameter of particles (microns)     0.50
>>> head['property'][2]
'Refractive index of particles '
>>> head['value'][2]
1.59
```

Reading from csv files

- The most common form of data file is simply a text file consisting of columns of data separated by spaces, tabs, etc.
- `read_table()` is exactly the same as `read_csv()` except it adds the keyword `sep`, which allows you to specify how columns of data are separated
- `read_table(sep=',')` is completely equivalent to `read_csv()`

Keyword	Value	Description
`sep`	`"\s+"`	space-delimited data
	`"\t"`	tab-delimited data
	str	string-delimited data
`delim_whitespace`	`True`	mixed tab/space-delimited data
`comment`	str	set comment symbol

Reading from text files

- Data about the planets stored in a text file with the data columns separated by spaces:

planetData.txt

```
planet  distance    mass  gravity diameter  year
Mercury     0.39   0.055     0.38     0.38  0.24
Venus       0.72    0.82     0.91     0.95  0.62
Earth       1.00    1.00     1.00     1.00  1.00
Mars        1.52    0.11     0.38     0.53  1.88
Jupiter     5.20     318     2.36     11.2  11.9
Saturn      9.58      95     0.92     9.45    29
Uranus      19.2      15     0.89     4.01    84
Neptune     30.0      17     1.12     3.88   164
Pluto       39.5  0.0024    0.071     0.19   248
```

Reading from text files


```
>>> planets = pd.read_table('planetData.txt', sep='\s+')
>>> planets
    planet  distance      mass  gravity  diameter    year
0  Mercury      0.39    0.0550    0.380      0.38    0.24
1    Venus      0.72    0.8200    0.910      0.95    0.62
2    Earth      1.00    1.0000    1.000      1.00    1.00
……
>>> planets = pd.read_table('planetData.txt', sep='\s+', index_col='planet')
>>> planets
         distance      mass  gravity  diameter    year
planet
Mercury      0.39    0.0550    0.380      0.38    0.24
Venus        0.72    0.8200    0.910      0.95    0.62
Earth        1.00    1.0000    1.000      1.00    1.00
……
>>> planets['distance']['Saturn']
9.58
```

Reading from Excel files

- **`Pandas`** can also read directly from Excel files (i.e., with .xls or .xlsx extensions)

`BloodPressure.xlsx`

	A	B	C	D	E
1	Date	Time	BP_sys	BP_dia	Pulse
2	1-Jun	23:33	119	70	71
3	2-Jun	5:57	129	83	59
4	2-Jun	22:19	113	67	59
5	3-Jun	5:24	131	77	55
6	3-Jun	23:19	114	65	60
7	4-Jun	6:54	119	75	55
8	4-Jun	21:40	121	68	56
9	5-Jun	6:29	130	83	56
10	5-Jun	22:16	113	61	67
11	6-Jun	5:23	116	81	60

Reading from Excel files


```
>>> bp = pd.read_excel('BloodPressure.xlsx', usecols="A:E")
>>> bp
         Date      Time  BP_sys  BP_dia  Pulse
0  2017-06-01  23:33:00     119      70     71
1  2017-06-02  05:57:00     129      83     59
2  2017-06-02  22:19:00     113      67     59
3  2017-06-03  05:24:00     131      77     55
4  2017-06-03  23:19:00     114      65     60
……
>>> bp = pd.read_excel('BloodPressure.xlsx', usecols="A:B,E")
>>> bp
         Date      Time  Pulse
0  2017-06-01  23:33:00     71
1  2017-06-02  05:57:00     59
2  2017-06-02  22:19:00     59
3  2017-06-03  05:24:00     55
4  2017-06-03  23:19:00     60
……
```

Dates and times in Pandas

- Pandas has special tools for handling dates and times. They make use of the Python library datetime, which defines, among other things, a useful datetime object:

```
>>> import datetime as dt
>>> t0 = dt.datetime.now()
>>> t0
datetime.datetime(2021, 5, 19, 16, 55, 20, 775949)
>>> t0.strftime('%Y-%m-%d')
'2021-05-19'
>>> t0.strftime('%m-%d-%Y')
'05-19-2021'
>>> t0.strftime('%d-%b-%Y')
'19-May-2021'
>>> t0.strftime('%H:%M:%S')
'16:55:20'
>>> t0.strftime('%Y-%B-%d %H:%M:%S')
'2021-May-19 16:55:20'
```

string format time
"strftime"

Code	Example	Code	Example
%a	Sun	%M	06
%A	Sunday	%-M	6
%w	0	%S	05
%d	08	%-S	5
%-d	8	%f	000000
%b	Sep	%z	+0000
%B	September	%Z	UTC
%m	09	%j	251
%-m	9	%-j	251
%y	13	%U	36
%Y	2013	%W	35
%H	07	%c	Sun Sep 8 07:06:05 2013
%-H	7	%x	09/08/13
%I	07	%X	07:06:05
%-I	7	%%	%
%p	AM	https://strftime.org/	

Dates and times in Pandas


```
>>> bp = pd.read_excel('BloodPressure.xlsx', usecols="A:E", parse_dates=[['Date', 'Time']])
>>> bp
             Date_Time  BP_sys  BP_dia  Pulse
0  2017-06-01 23:33:00     119      70     71
1  2017-06-02 05:57:00     129      83     59
2  2017-06-02 22:19:00     113      67     59
3  2017-06-03 05:24:00     131      77     55
4  2017-06-03 23:19:00     114      65     60
5  2017-06-04 06:54:00     119      75     55
.....
```

BloodPressure.xlsx

	A	B	C	D	E
1	Date	Time	BP_sys	BP_dia	Pulse
2	1-Jun	23:33	119	70	71
3	2-Jun	5:57	129	83	59
4	2-Jun	22:19	113	67	59
5	3-Jun	5:24	131	77	55
6	3-Jun	23:19	114	65	60
7	4-Jun	6:54	119	75	55
8	4-Jun	21:40	121	68	56
9	5-Jun	6:29	130	83	56
10	5-Jun	22:16	113	61	67
11	6-Jun	5:23	116	81	60

The `parse_dates` keyword argument can also be used with the `read_csv` and `read_table` methods

Series

- Pandas has two principal data structures: Series and DataFrame, which form the basis for most activities using Pandas. Both Series and DataFrame use NumPy array extensively, but allow more versatile ways of indexing.
- A Series object is a one-dimensional DataFrame.
- Series is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.).
- The axis labels are collectively referred to as the index.

```
>>> s = pd.Series(data, index=None)
```

Here, s is a Pandas Series, data can be a Python dict, a ndarray, or a scalar value (like 5). The passed index is a list of axis labels.

- Both integer and label-based indexing are supported. If the index is not provided, then the index will default to range(n) where n is the length of data.

Series

SUSTech Southern University of Science and Technology

- Create `Series` from ndarrays:

```
>>> import numpy as np
>>> import pandas as pd
>>> s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
>>> type(s)
<class 'pandas.core.series.Series'>
>>> s
a -0.367740
b 0.855453
c -0.518004
d -0.060861
e -0.277982
dtype: float64
>>> s.index
Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
>>> s.values
array([-0.367740, 0.855453, -0.518004, -0.060861, -0.277982])
>>> pd.Series(np.random.randn(5))
0 0.334947
1 -2.184006
2 -0.209440
3 -0.492398
4 -1.507088
dtype: float64
```

index

class

- The `Pandas Series` function can turn a list, dictionary, or `NumPy` array into a `Pandas Series`:

```
>>> ht = pd.Series([160.0-4.9*t*t for t in range(6)])
>>> ht
0    160.0
1    155.1
2    140.4
3    115.9
4     81.6
5     37.5
dtype: float64
```

index

value

list comprehension

- You can specify axis labels for `index`, i.e., `index=['a', 'b', 'c', 'd', 'e']`.
- When data is a `ndarray`, the `index` must be the same length as data. In series `s`, by default the type of values of all the elements is `dtype: float64`.
- You can find out the index for a series using `index` attribute. The `values` attribute returns a `ndarray` containing only values, while the axis labels are removed.
- If no labels for the index is passed, one will be created having a range of index values `[0,..., len(data) - 1]`.

Series


```
>>> ht[2]
140.4
>>> ht[1:4]
1    155.1
2    140.4
3    115.9
dtype: float64
>>> ht.values
array([160. , 155.1, 140.4, 115.9,
81.6,  37.5])
>>> ht.index
RangeIndex(start=0, stop=6, step=1)
```

```
>>> heights = pd.Series([188, 157, 173,
169, 155], index=['Jake', 'Sarah',
'Heather', 'Chris', 'Alex'])
>>> heights
Jake       188
Sarah      157
Heather    173
Chris      169
Alex       155
dtype: int64
>>> heights.values
array([188, 157, 173, 169, 155],
dtype=int64)
>>> heights.index
Index(['Jake', 'Sarah', 'Heather', 'Chris',
'Alex'], dtype='object')
```

Series

- Create `Series` from Dictionaries:

```
>>> heights = pd.Series([188, 157, 173,
169, 155], index=['Jake', 'Sarah',
'Heather', 'Chris', 'Alex'])
>>> htd = heights.to_dict()
>>> htd
{'Jake': 188, 'Sarah': 157, 'Heather':
173, 'Chris': 169, 'Alex': 155}
>>> pd.Series(htd)
Jake       188
Sarah      157
Heather    173
Chris      169
Alex       155
dtype: int64
```

```
>>> import numpy as np
>>> import pandas as pd
>>> d = {'a' : 0., 'b' : 1., 'c' : 2.}
>>> pd.Series(d)
a 0.0
b 1.0
c 2.0
dtype: float64
>>> pd.Series(d, index=['b', 'c', 'd', 'a'])
b 1.0
c 2.0
d NaN
a 0.0
dtype: float64
```

- When a series is created using dictionaries, by default the keys will be `index` labels.
- While creating a series using a dictionary, if labels are passed for the `index`, the values corresponding to the labels in the index will be pulled out. The order of index labels will be preserved.
- If a value is not associated for a label, then `NaN` is printed. `NaN` (not a number) is the standard missing data marker used in `pandas`.

Series

- Create `Series` from Scalar data:

```
>>> import numpy as np
>>> import pandas as pd
>>> pd.Series(5., index=['a', 'b', 'c', 'd', 'e'])
a 5.0
b 5.0
c 5.0
d 5.0
e 5.0
dtype: float64
```

- A `Pandas Series` created from scalar value.
- If data is a scalar value, an `index` must be provided.
- The value will be repeated to match the length of the `index`.

- You can provide index or slice data by index numbers in a `Pandas Series`.
- You can also specify a Boolean array indexing for `Pandas Series`.
- Multiple indices are specified as a list. The `index` can be an integer value or a label.
- Check for the presence of a label in `Series` using `in` operator.

- Indexing and slicing `Series`:

```
>>> s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
>>> s
a 0.481557
b 2.053330
c -1.799993
d -0.396880
e -1.270751
dtype: float64
>>> s[0]
0.48155677569897515
>>> s[1:3]
b 2.053330
c -1.799993
dtype: float64
>>> s[:3]
a 0.481557
b 2.053330
c -1.799993
dtype: float64
>>> s[s > .5]
b 2.05333
dtype: float64
>>> s[[4, 3, 1]]
e -1.270751
d -0.396880
b 2.053330
dtype: float64
>>> s['a']
0.48155677569897515
>>> s[['a','c']]
a -1.077452
c 1.418233
dtype: float64
>>> 'e' in s
True
>>> 'f' in s
False
```

Time series

- One of the most common uses of `Pandas series` involves a time series in which the series is indexed by timestamps:

```
>>> ht = pd.Series([160.0-4.9*t*t for t in range(6)])
>>> dtr = pd.date_range('2017-07-22', periods=6)
>>> dtr
DatetimeIndex(['2017-07-22', '2017-07-23', '2017-07-24',
               '2017-07-25', '2017-07-26', '2017-07-27'],
              dtype='datetime64[ns]', freq='D')
>>> ht.index = dtr
>>> ht
2017-07-22    160.0
2017-07-23    155.1
2017-07-24    140.4
2017-07-25    115.9
2017-07-26     81.6
2017-07-27     37.5
Freq: D, dtype: float64
```

DataFrame

- `DataFrame` is a two-dimensional, labeled data structure with columns of potentially different types.
- You can think of it like a spreadsheet or database table, or a `dict` of `Series` objects.

```
>>> bp = pd.read_excel('BloodPressure.xlsx', usecols="A:E",
parse_dates=[['Date', 'Time']])
>>> bp.head(3)
            Date_Time  BP_sys  BP_dia  Pulse
0 2017-06-01 23:33:00     119      70     71
1 2017-06-02 05:57:00     129      83     59
2 2017-06-02 22:19:00     113      67     59
>>> bp.tail(4)
             Date_Time  BP_sys  BP_dia  Pulse
44 2017-07-15 22:57:00     109      63     62
45 2017-07-16 06:45:00     124      78     47
46 2017-07-16 22:15:00     121      74     58
47 2017-07-17 06:22:00     113      79     57
```

BloodPressure.xlsx

	A	B	C	D	E
1	Date	Time	BP_sys	BP_dia	Pulse
2	1-Jun	23:33	119	70	71
3	2-Jun	5:57	129	83	59
4	2-Jun	22:19	113	67	59
5	3-Jun	5:24	131	77	55
6	3-Jun	23:19	114	65	60
7	4-Jun	6:54	119	75	55
8	4-Jun	21:40	121	68	56
9	5-Jun	6:29	130	83	56
10	5-Jun	22:16	113	61	67
11	6-Jun	5:23	116	81	60

DataFrame

- The recommended scheme of indexing `DataFrame` is to use the `iloc` and `loc` methods, which are faster and more versatile
- The `iloc` method indexes the `DataFrame` by **row and column number**:

```
>>> bp.iloc[0, 2]
70
>>> bp.iloc[1, 0:3]
Date_Time 2017-06-02 05:57:00
BP_sys                      129
BP_dia                       83
Name: 1, dtype: object
>>> bp = bp.set_index('Date_Time')
>>> bp.head(2)
                     BP_sys  BP_dia  Pulse
Date_Time
2017-06-01 23:33:00     119      70     71
2017-06-02 05:57:00     129      83     59
>>> bp.iloc[1, 0:2]
BP_sys    129
BP_dia     83
Pulse      59
Name: 2017-06-02 05:57:00, dtype: int64
```

BloodPressure.xlsx

	A	B	C	D	E
1	Date	Time	BP_sys	BP_dia	Pulse
2	1-Jun	23:33	119	70	71
3	2-Jun	5:57	129	83	59
4	2-Jun	22:19	113	67	59
5	3-Jun	5:24	131	77	55
6	3-Jun	23:19	114	65	60
7	4-Jun	6:54	119	75	55
8	4-Jun	21:40	121	68	56
9	5-Jun	6:29	130	83	56
10	5-Jun	22:16	113	61	67
11	6-Jun	5:23	116	81	60

DataFrame

- The `loc` method is an extremely versatile tool for indexing DataFrames. It can select data based on conditions:

```
>>> PulseAM = bp.loc[bp.index.hour<12, 'Pulse']
>>> PulseAM
Date_Time
2017-06-02 05:57:00    59
2017-06-03 05:24:00    55
……
Name: Pulse, dtype: int64
>>> PulsePM = bp.loc[bp.index.hour>=12, 'Pulse']
>>> PulsePM
Date_Time
2017-06-01 23:33:00    71
2017-06-02 22:19:00    59
……
Name: Pulse, dtype: int64
```

DataFrame


```
>>> planets = pd.read_table('planetData.txt', sep='\s+', index_col='planet')
>>> planets.loc[(planets['mass'] > 1.0) & (planets['gravity'] < 1.0)]
        distance  mass  gravity  diameter  year
planet
Saturn      9.58  95.0     0.92      9.45  29.0
Uranus     19.20  15.0     0.89      4.01  84.0
>>> planets.loc[(planets['mass'] > 1.0) & (planets['gravity'] < 1.0),
                'mass':'gravity']
        mass  gravity
planet
Saturn  95.0     0.92
Uranus  15.0     0.89
>>> planets.loc[(planets.mass > 1.0) & (planets.gravity < 1.0)]
        distance  mass  gravity  diameter  year
planet
Saturn      9.58  95.0     0.92      9.45  29.0
Uranus     19.20  15.0     0.89      4.01  84.0
```

planetData.txt

planet	distance	mass	gravity	diameter	year
Mercury	0.39	0.055	0.38	0.38	0.24
Venus	0.72	0.82	0.91	0.95	0.62
Earth	1.00	1.00	1.00	1.00	1.00
Mars	1.52	0.11	0.38	0.53	1.88
Jupiter	5.20	318	2.36	11.2	11.9
Saturn	9.58	95	0.92	9.45	29
Uranus	19.2	15	0.89	4.01	84
Neptune	30.0	17	1.12	3.88	164
Pluto	39.5	0.0024	0.071	0.19	248

Create a DataFrame

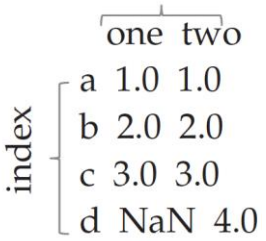
- `DataFrame` accepts many different kinds of input like `Dict` of one-dimensional `ndarrays`, `lists`, `dicts`, or `Series`, two-dimensional `ndarrays`, structured or record `ndarray`, a dictionary of `Series`, or another `DataFrame`.
- `df = pd.DataFrame(data=None, index=None, columns=None)`
- Here, `df` is the `DataFrame` and data can be `NumPy ndarray`, `dict`, or `DataFrame`.
- Along with the data, you can optionally pass an `index` (row labels) and `columns` (column labels) attributes as arguments.
- Both `index` and `columns` will default to `range(n)` where `n` is the length of data, if they are not provided.
- When the `data` is a dictionary and `columns` are not specified, then the `DataFrame` column labels will be dictionary's `keys`.

Create a DataFrame

- A `DataFrame` can be created from a Dictionary of Series/Dictionaries:

```
>>> import pandas as pd
>>> dict_series = {'one' : pd.Series([1., 2., 3.], index=['a', 'b', 'c']),
                   'two' : pd.Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd'])}
>>> df = pd.DataFrame(dict_series)
>>> df.shape
(4, 2)
>>> df.index
Index(['a', 'b', 'c', 'd'], dtype='object')
>>> df.columns
Index(['one', 'two'], dtype='object')
>>> list(df.columns)
['one', 'two']

>>> dicts_only = {'a':[1,2,3], 'b':[4,5,6]}
>>> dict_df = pd.DataFrame(dicts_only)
>>> dict_df
   a b
0  1 4
1  2 5
2  3 6
>>> dict_df.index
RangeIndex(start=0, stop=3, step=1)
```



- If the number of labels specified in the various series are not the same, then the resulting index will be the union of all the index labels of various series.
- Get the index labels for the DataFrame using `index` attribute. With `columns` attribute, you get all the columns of the DataFrame.

Create a DataFrame

- A `DataFrame` can be created using the `Pandas DataFrame` routine based on list-like objects such as `list`, `NumPy array`, or `dictionary`:

```
>>> optmat = {'mat': ['silica', 'titania', 'PMMA', 'PS'], 'index': [1.46, 2.40, 1.49, 1.59],
'density': [2.03, 4.2, 1.19, 1.05]}
>>> omdf = pd.DataFrame(optmat)
>>> omdf
       mat  index  density
0   silica   1.46     2.03
1  titania   2.40     4.20
2     PMMA   1.49     1.19
3       PS   1.59     1.05
```

- The column order can be changed:

```
>>> omdf = pd.DataFrame(optmat, columns=['index', 'mat', 'density'])
>>> omdf
   index      mat  density
0   1.46   silica     2.03
1   2.40  titania     4.20
2   1.49     PMMA     1.19
3   1.59       PS     1.05
```

Create a DataFrame

- We can also create a `DataFrame` with empty columns and fill in the data later:

```
>>> omdf1 = pd.DataFrame(index=['silica', 'titania', 'PMMA', 'PS'], columns={'density', 'index'})
>>> omdf1
        density index
silica      NaN   NaN
titania     NaN   NaN
PMMA        NaN   NaN
PS          NaN   NaN
>>> omdf1.dtypes
density    object
index      object
dtype: object
```

`NaN`: not-a-number

- The index and density can be changed to `float` data type:

```
>>> omdf1[['index', 'density']] = omdf1[['index', 'density']].apply(pd.to_numeric)
>>> omdf1.dtypes
density    float64
index      float64
dtype: object
```

Extract information from a DataFrame

- The information in a `DataFrame` can be extracted and summarized in a variety of ways using the tools of `Pandas`:

```
>>> planets = pd.read_table('planetData.txt', sep='\s+', index_col='planet')
>>> planets
         distance      mass  gravity  diameter    year
planet
Mercury      0.39    0.0550    0.380      0.38    0.24
Venus        0.72    0.8200    0.910      0.95    0.62
......
>>> planets.sort_values(by='mass')
         distance      mass  gravity  diameter    year
planet
Pluto       39.50    0.0024    0.071      0.19  248.00
Mercury      0.39    0.0550    0.380      0.38    0.24
Mars         1.52    0.1100    0.380      0.53    1.88
......
>>> planets.sort_values(by='mass', ascending=False)
         distance      mass  gravity  diameter    year
planet
Jupiter      5.20  318.0000    2.360     11.20   11.90
Saturn       9.58   95.0000    0.920      9.45   29.00
Neptune     30.00   17.0000    1.120      3.88  164.00
......
```

```
planets.sort_values(by='mass', inplace=True)
planets.head()
```

planet	distance	mass	gravity	diameter	year
Pluto	39.50	0.0024	0.071	0.19	248.00
Mercury	0.39	0.0550	0.380	0.38	0.24
Mars	1.52	0.1100	0.380	0.53	1.88
Venus	0.72	0.8200	0.910	0.95	0.62
Earth	1.00	1.0000	1.000	1.00	1.00

Extract information from a DataFrame

- Conditional indexing:

```
>>> planets[planets['gravity']>1]
         distance   mass  gravity  diameter   year
planet
Jupiter       5.2  318.0     2.36     11.20   11.9
Neptune      30.0   17.0     1.12      3.88  164.0
>>> planets[planets.gravity>1]
         distance   mass  gravity  diameter   year
planet
Jupiter       5.2  318.0     2.36     11.20   11.9
Neptune      30.0   17.0     1.12      3.88  164.0
>>> planets['gravity']>1
planet
Mercury    False
Venus      False
Earth      False
Mars       False
Jupiter     True
Saturn     False
Uranus     False
Neptune     True
Pluto      False
Name: gravity, dtype: bool
```

planetData.txt

planet	distance	mass	gravity	diameter	year
Mercury	0.39	0.055	0.38	0.38	0.24
Venus	0.72	0.82	0.91	0.95	0.62
Earth	1.00	1.00	1.00	1.00	1.00
Mars	1.52	0.11	0.38	0.53	1.88
Jupiter	5.20	318	2.36	11.2	11.9
Saturn	9.58	95	0.92	9.45	29
Uranus	19.2	15	0.89	4.01	84
Neptune	30.0	17	1.12	3.88	164
Pluto	39.5	0.0024	0.071	0.19	248

Extract information from a DataFrame

- Add a "volume" column for each planet using $V = 1/6*\pi d^3$:

```
>>> planets['volume'] = np.pi * planets['diameter']**3 / 6.0
>>> planets
         distance      mass  gravity  diameter    year      volume
planet
Mercury      0.39    0.0550    0.380      0.38    0.24    0.028731
Venus        0.72    0.8200    0.910      0.95    0.62    0.448921
Earth        1.00    1.0000    1.000      1.00    1.00    0.523599
Mars         1.52    0.1100    0.380      0.53    1.88    0.077952
......
>>> planets['volume'] = planets['volume'] / planets.volume.Earth
>>> planets
         distance      mass  gravity  diameter    year       volume
planet
Mercury      0.39    0.0550    0.380      0.38    0.24     0.054872
Venus        0.72    0.8200    0.910      0.95    0.62     0.857375
Earth        1.00    1.0000    1.000      1.00    1.00     1.000000
Mars         1.52    0.1100    0.380      0.53    1.88     0.148877
```

Extract information from a DataFrame

SUSTech Southern University of Science and Technology

```
>>> bp = pd.read_excel('BloodPressure.xlsx', usecols="A:E", parse_dates=[['Date', 'Time']])
>>> bp = bp.set_index('Date_Time')
>>> bp
                     BP_sys  BP_dia  Pulse
Date_Time
2017-06-01 23:33:00     119      70     71
2017-06-02 05:57:00     129      83     59
......
>>> bp['BP_sys'].mean()
119.27083333333333
>>> bp.BP_sys.mean()
119.27083333333333
>>> bp['BP_sys'].max()
131
>>> bp['BP_sys'].min()
105
>>> bp['BP_sys'].count()
48
>>> bp.index.min()
Timestamp('2017-06-01 23:33:00')
>>> bp.index.max()
Timestamp('2017-07-17 06:22:00')
>>> bp.index.max()-bp.index.min()
Timedelta('45 days 06:49:00')
```

count(): number of non-null entries

```
bp = pd.read_excel('BloodPressure.xlsx', usecols="A:E", parse_dates=[['Date', 'Time']])
bp.head()
```

	Date_Time	BP_sys	BP_dia	Pulse
0	2017-06-01 23:33:00	119	70	71
1	2017-06-02 05:57:00	129	83	59
2	2017-06-02 22:19:00	113	67	59
3	2017-06-03 05:24:00	131	77	55
4	2017-06-03 23:19:00	114	65	60

```
>>> help(bp.index.max)
max(axis=None, skipna=True, *args, **kwargs) method of
pandas.core.indexes.datetimes.DatetimeIndex instance
    Return the maximum value of the Index.
Parameters
    ----------
    axis : int, optional
        For compatibility with NumPy. Only 0 or None are
allowed.
    skipna : bool, default True
        Exclude NA/null values when showing the result.
>>> idx = pd.Index([3, 2, 1])
>>> idx.max()
    3
>>> idx = pd.Index(['c', 'b', 'a'])
>>> idx.max()
    'c'
```

Extract information from a DataFrame

- Check if there is a systematic difference in the blood pressure and pulse readings in the morning and the evening:

```
>>> PulseAM = bp.loc[bp.index.hour<12, 'Pulse']
>>> PulsePM = bp.loc[bp.index.hour>=12, 'Pulse']
>>> PulseAM.mean(), PulseAM.std(), PulseAM.sem()
(57.58620689655172, 5.791109204013328 5, 1.0753819820594928)
>>> PulsePM.mean(), PulsePM.std(), PulsePM.sem()
(61.78947368421053, 4.939398831711553, 1.133175807856501)
>>> from scipy.stats import ttest_ind
>>> ttest_ind(PulseAM, PulsePM)
Ttest_indResult(statistic=-2.6017534012734376, pvalue=0.012436030339416216)
```

`sem`: standard error of mean

`ttest_ind`: an independent two sample t-test

Grouping and aggregation

- `Pandas` allows to group data and analyze the subgroups in useful and powerful ways:
- Example data: all airplane departures from Newark Liberty International Airport (EWR) on a particular (stormy) day:

ewrFlights20180516.csv

	A	B	C	D	E	F	G	H	I	J
1	Destination	Airline	Flight	Departure	Terminal	Status	Arrival_time	A_day	Scheduled	S_day
2	Baltimore (BWI)	Southwest Airlines	WN 8512	12:09 AM		Landed				
3	Baltimore (BWI)	Mountain Air Cargo	C2 7304	12:10 AM		Unknown				
4	Paris (ORY)	Norwegian Air Shuttle	DY 7192	12:30 AM	B	Landed	1:48 PM		1:35 PM	
5	Paris (ORY)	euroAtlantic Airways	YU 7192	12:30 AM	B	Landed	1:48 PM		1:35 PM	
6	Rockford (RFD)	UPS	5X 108	12:48 AM		Unknown				
7	Los Angeles (LAX)	FedEx	FX 1026	1:15 AM		Landed - On-time	4:07 AM			
8	Hong Kong (HKG)	American Airlines	AA 8942	1:55 AM	B	Landed - On-time	5:01 AM		5:30 AM	
9	Hong Kong (HKG)	Cathay Pacific	CX 899	1:55 AM	B	Landed - On-time	5:01 AM		5:30 AM	
10	Baltimore (BWI)	Mountain Air Cargo	C2 8308	3:20 AM		Landed - On-time	4:47 AM			
11	Atlanta (ATL)	FedEx	FX 1988	3:25 AM		Landed - Delayed	5:45 AM			
12	Orlando (MCO)	FedEx	FX 1966	3:25 AM		Landed - On-time	5:50 AM			
13	Detroit (DTW)	FedEx	FX 1982	3:30 AM		Landed - On-time	5:14 AM			

Grouping and aggregation


```
>>> ewr = pd.read_csv('ewrFlights20180516.csv')
>>> ewr.head()
       Destination                 Airline  ... Scheduled S_day
0  Baltimore (BWI)     Southwest Airlines  ...       NaN   NaN
1  Baltimore (BWI)     Mountain Air Cargo  ...       NaN   NaN
2      Paris (ORY)  Norwegian Air Shuttle  ...   1:35 PM   NaN
3      Paris (ORY)   euroAtlantic Airways  ...   1:35 PM   NaN
4   Rockford (RFD)                    UPS  ...       NaN   NaN

[5 rows x 10 columns]
>>> ewr.tail()
            Destination                 Airline  ... Scheduled S_day
1550    Louisville (SDF)                     UPS  ...       NaN   NaN
1551  Indianapolis (IND)                  FedEx  ...       NaN   NaN
1552          Rome (FCO)  Norwegian Air Shuttle  ...   1:40 PM   1.0
1553        Athens (ATH)               Emirates  ...   4:05 PM   1.0
1554        Athens (ATH)        JetBlue Airways  ...   4:05 PM   1.0
```

```
ewr = pd.read_csv('ewrFlights20180516.csv')
ewr.head()
```

	Destination	Airline	Flight	Departure	Terminal	Status	Arrival_time	A_day	Scheduled	S_day
0	Baltimore (BWI)	Southwest Airlines	WN 8512	12:09 AM	NaN	Landed	NaN	NaN	NaN	NaN
1	Baltimore (BWI)	Mountain Air Cargo	C2 7304	12:10 AM	NaN	Unknown	NaN	NaN	NaN	NaN
2	Paris (ORY)	Norwegian Air Shuttle	DY 7192	12:30 AM	B	Landed	1:48 PM	NaN	1:35 PM	NaN
3	Paris (ORY)	euroAtlantic Airways	YU 7192	12:30 AM	B	Landed	1:48 PM	NaN	1:35 PM	NaN
4	Rockford (RFD)	UPS	5X 108	12:48 AM	NaN	Unknown	NaN	NaN	NaN	NaN

Grouping and aggregation

- The `value_counts()` method finds all the unique entries in a `Series` (or `DataFrame` column) and reports the number of times each entry appears:

```
>>> ewr['Status'].value_counts()
Landed - On-time        757
Landed - Delayed        720
Canceled                 41
Landed                   18
En Route - Delayed       10
Unknown                   4
Scheduled - Delayed       2
En Route - On-time        1
En Route                  1
Diverted                  1
Name: Status, dtype: int64
>>> ewr['Status'].value_counts().sum()
1555
>>> ewr['Terminal'].value_counts()
C    826
A    471
B    191
```

```
ewr = pd.read_csv('ewrFlights20180516.csv')
ewr.head()
```

	Destination	Airline	Flight	Departure	Terminal	Status	Arrival_time	A_day	Scheduled	S_day
0	Baltimore (BWI)	Southwest Airlines	WN 8512	12:09 AM	NaN	Landed	NaN	NaN	NaN	NaN
1	Baltimore (BWI)	Mountain Air Cargo	C2 7304	12:10 AM	NaN	Unknown	NaN	NaN	NaN	NaN
2	Paris (ORY)	Norwegian Air Shuttle	DY 7192	12:30 AM	B	Landed	1:48 PM	NaN	1:35 PM	NaN
3	Paris (ORY)	euroAtlantic Airways	YU 7192	12:30 AM	B	Landed	1:48 PM	NaN	1:35 PM	NaN
4	Rockford (RFD)	UPS	5X 108	12:48 AM	NaN	Unknown	NaN	NaN	NaN	NaN

groupby method

- `groupby` method can obtain the status of each flight broken down by terminal:

```
>>> ewr['Status'].groupby(ewr['Terminal']).value_counts()
Terminal  Status
A         Landed - On-time       229
          Landed - Delayed       218
          Canceled                21
          Landed                   3
B         Landed - On-time       104
          Landed - Delayed        70
          En Route - Delayed       6
          Canceled                 4
          Landed                   4
          Scheduled - Delayed      2
          En Route - On-time       1
C         Landed - Delayed       413
          Landed - On-time       395
          Canceled                14
          En Route - Delayed       4
Name: Status, dtype: int64
```

Iterating over groups

- Sometimes it is useful to iterate over groups to perform a calculation:

```
>>> for airln, grp in ewr.groupby(ewr['Airline']):
...     print(f'\nairln = {airln}: \ngrp:')
...     print(grp)
...

airln = ANA:
grp:
             Destination Airline   Flight Departure  ... Arrival_time A_day Scheduled  S_day
134   San Francisco (SFO)     ANA  NH 7007   7:00 AM  ...     11:13 AM   NaN  10:18 AM    NaN
189     Los Angeles (LAX)     ANA  NH 7229   7:59 AM  ...     10:57 AM   NaN  11:05 AM    NaN
303         Chicago (ORD)     ANA  NH 7469   8:59 AM  ...     10:39 AM   NaN  10:25 AM    NaN
438           Tokyo (NRT)     ANA  NH 6453  11:00 AM  ...      1:20 PM   NaN   1:55 PM    NaN
562         Chicago (ORD)     ANA  NH 7569   1:20 PM  ...      3:16 PM   NaN   2:44 PM    NaN
1140    Los Angeles (LAX)     ANA  NH 7235   6:43 PM  ...      9:54 PM   NaN   9:41 PM    NaN
1533      Sao Paulo (GRU)     ANA  NH 7214  10:05 PM  ...     10:06 AM   1.0   8:50 AM    1.0

[7 rows x 10 columns]

airln = AVIANCA:
grp:
            Destination  Airline   Flight Departure  ... Arrival_time A_day Scheduled  S_day
81         Dulles (IAD)  AVIANCA  AV 2135   6:05 AM  ...      7:17 AM   NaN   7:25 AM    NaN
367        Dulles (IAD)  AVIANCA  AV 2233  10:00 AM  ...     11:10 AM   NaN  11:20 AM    NaN
422         Miami (MIA)  AVIANCA  AV 2002  10:44 AM  ...      1:30 PM   NaN   1:46 PM    NaN
805  San Salvador (SAL)  AVIANCA   AV 399   3:55 PM  ...          NaN   NaN   7:05 PM    NaN
890        Bogota (BOG)  AVIANCA  AV 2245   4:45 PM  ...     12:42 AM   1.0   9:35 PM    NaN

[5 rows x 10 columns]
……
```

Iterating over groups

- Find airlines that landed 12 or more flights:

```
>>> ot = []
>>> for airln, grp in ewr.groupby(ewr['Airline']):
...     ontime = grp.Status[grp.Status == 'Landed - On-time'].count()
...     delayd = grp.Status[grp.Status == 'Landed - Delayed'].count()
...     totl = ontime+delayd
...     if totl >= 12:
...         ot.append([airln, totl, ontime/totl])
...
>>> ot[0:3]
[['Air Canada', 129, 0.4728682170542636],
 ['Air China', 24, 0.75],
 ['Air New Zealand', 34, 0.6176470588235294]]
>>> t = pd.DataFrame.from_records(ot, columns=['Airline', 'Flights Landed', 'On-time fraction'])
>>> t.sort_values(by='On-time fraction', ascending=False)
               Airline  Flights Landed  On-time fraction
1            Air China              24          0.750000
2      Air New Zealand              34          0.617647
20     Virgin Atlantic              26          0.615385
9      Delta Air Lines              33          0.606061
14   Republic Airlines              66          0.606061
4    American Airlines              27          0.592593
……
```

Exercises

Exercise 1

- Create a script to count how many days passed from a given date, 2000.12.31.

Hint: You can get the date using the `datetime` package

Exercise 2

- Create two `2x5` and `5x2` DataFrames using the following lists:

  ```
  a = [1,2,3,4,5]
  b = [6,7,8,9,10]
  ```

  with row or column names `['a', 'b']` and `['A', 'B', 'C', 'D', 'E']`

Exercise 3

- You have collected a list of data science and algorithm marks from course assignments:

```
data = [['DS', 'Linked_list', 10],
        ['DS', 'Stack', 9],
        ['DS', 'Queue', 7],
        ['Algo', 'Greedy', 8],
        ['Algo', 'DP', 6],
        ['Algo', 'BackTrack', 5], ]
```

with column names `['Course', 'Session', 'Marks']`

Calculate the mean mark for each course using `Pandas`.

Exercise 4

- Find out which male students failed the exam by filtering the scores using `Pandas`, the data has been organized into a dictionary:

```
records = { 'Name': ['Mary', 'Maria', 'Anna', 'John', 'Jake', 'Joe' ],
            'Gender': ['Female', 'Female', 'Female', 'Male', 'Male', 'Male'],
            'Score': [88, 92, 56, 70, 55, 48] }
```

Exercise 5

- Find out the top five busiest destinations evaluated by total flight numbers or total unique airline numbers:

ewrFlights20180516.csv

	A	B	C	D	E	F	G	H	I	J
1	Destination	Airline	Flight	Departure	Terminal	Status	Arrival_time	A_day	Scheduled	S_day
2	Baltimore (BWI)	Southwest Airlines	WN 8512	12:09 AM		Landed				
3	Baltimore (BWI)	Mountain Air Cargo	C2 7304	12:10 AM		Unknown				
4	Paris (ORY)	Norwegian Air Shuttle	DY 7192	12:30 AM	B	Landed	1:48 PM		1:35 PM	
5	Paris (ORY)	euroAtlantic Airways	YU 7192	12:30 AM	B	Landed	1:48 PM		1:35 PM	
6	Rockford (RFD)	UPS	5X 108	12:48 AM		Unknown				
7	Los Angeles (LAX)	FedEx	FX 1026	1:15 AM		Landed - On-time	4:07 AM			
8	Hong Kong (HKG)	American Airlines	AA 8942	1:55 AM	B	Landed - On-time	5:01 AM		5:30 AM	
9	Hong Kong (HKG)	Cathay Pacific	CX 899	1:55 AM	B	Landed - On-time	5:01 AM		5:30 AM	
10	Baltimore (BWI)	Mountain Air Cargo	C2 8308	3:20 AM		Landed - On-time	4:47 AM			
11	Atlanta (ATL)	FedEx	FX 1988	3:25 AM		Landed - Delayed	5:45 AM			
12	Orlando (MCO)	FedEx	FX 1966	3:25 AM		Landed - On-time	5:50 AM			
13	Detroit (DTW)	FedEx	FX 1982	3:30 AM		Landed - On-time	5:14 AM			

https://github.com/djpine/python-scieng-public/blob/master/chap10/ewrFlights20180516.csv