Pandas

Data Science 2 / Data & AI 3

Karel de Grote Hogeschool

Reference: J. VanderPlas - O'Reilly 2022 (second edition)

Revision

Revision - Indexing

- What is the proper indexing to retrieve the the values in the yellow squares?

- The blue squares?

- The red squares?

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25
26	27	28	29	30

Revision

- What is the type of this numpy array?

```
X= np.array(
    [[1,2,3],
    [4,"5",6],
    [7,8,9]
  ])
```

Revision

- How to replace items that satisfy a condition without affecting the original array?

```
The input is:
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])


The expected output is:
array([ 0, -1,  2, -1,  4, -1,  6, -1,  8, -1])
```

Agenda

1. Introduction to Pandas
2. Indexing and Selection

   Bis: Reading files
3. Operations and Missing Values
4. Merge and Join
5. Aggregation and Grouping
6. Working with Strings

1

Introduction to Pandas

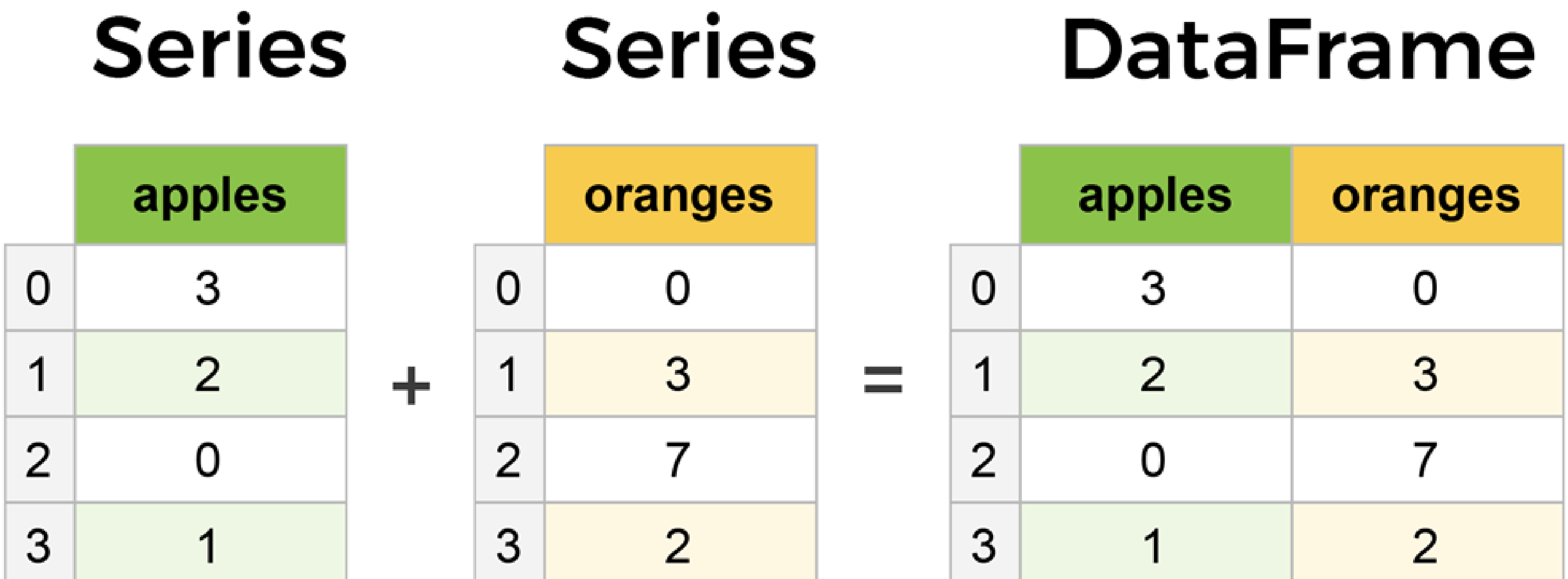
What is Pandas

Python library with flexible data structures developed for Data Scientists

DataFrame

Series

Data Structures are build on Numpy arrays



source: https://www.learndatasci.com/tutorials/python-pandas-tutorial-complete-introduction-for-beginners/

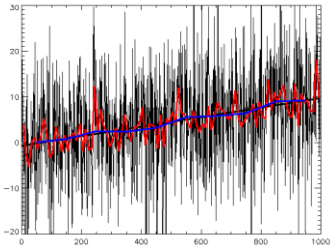
What is Pandas

Importing exporting and processing multiple data sources

Uniform handling missing data

N.A.

Explicitly defined indexes enabling advanced indexing, slicing and subsetting



Time series functionality

Advanced data manipulation

- GroupBy
- Joining
- ...

Pandas Series

Series as generalized NumPy array

- Numpy array: implicitly defined integer index
- Pandas Series: explicitly defined index

```
import pandas as pd
data = pd.Series([0.25, 0.5, 0.75, 1.0], index=[2, 5, 3, 7])
data[5]                                  # 0.5
data.values
data.index
data.dtype
```

```
2    0.25
5    0.50
3    0.75
7    1.00
dtype: float64
```

Series as specialized dictionary

- Python dictionary: values can have different types
- Pandas Series: all values have the same type (efficiency!)

```
population = pd.Series({'be': 10, 'nl': 8})
data['be']                               # 10
```

```
be    10
nl     8
dtype: int64
```

Pandas Dataframes

Dataframe as generalized 2D NumPy array

```
countries = pd.DataFrame( [ [11.7, 30688], [17.7, 41850] ] ,
                          columns=['population', 'area']))
countries
countries.info()
countries.describe()          # gives statistics of all the columns
```

	population	area
0	11.7	30688
1	17.7	41850

Dataframe as specialized dictionary

```
population = pd.Series({'be': 11.7, 'nl': 17.7})
area = pd.Series({'be': 30688, 'nl': 41850})
countries = pd.DataFrame({'population': population, 'area': area})
countries['area']          # or countries.area
```

	population	area
be	11.7	30688
nl	17.7	41850

2

Data Indexing and Selection

Indexing and Selection

1. Data Selection in Series
   - Series as dictionary
   - Series as one-dimensional array
   - Indexers: loc, iloc, and ix
      - Avoid *ix* because it is no longer available in modern pandas versions

2. Data Selection in DataFrame
   - DataFrame as a dictionary
   - DataFrame as two-dimensional array
   - Additional indexing conventions

Indexing and Selection - Series

```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])

# explicit index: .loc
data.loc[1]     # 'a'
# slicing
data.loc[1:3]   # 1 a
                # 3 b, explicit index: final index is included

# implicit index: .iloc
data.iloc[1]    # 'b'
# slicing
data.iloc[1:3] # 3 b
                # 5 c, implicit index: final index is excluded

# masking and fancy indexing
data[(data == 'a') | (data == 'b')]
data.loc[[1,3]]
```

1	a
3	b
5	c

Indexing and Selection - Dataframe

```
data= pd.DataFrame([ {'popul': 11.7, 'area': 30688},
                     {'popul': 17.7, 'area': 41850}],
                     index=['be', 'nl'])
# add new column
data['density'] = data['popul'] / data['area']
```

	population	area	density
be	11.7	30688	0.000381
nl	17.7	41850	0.000423

```
# implicit index: .iloc
data.iloc[:1, :1]    # implicit index: final index is excluded
                     # -> 1x1

# explicit index: .loc
data.loc[: 'nl ', : 'area']  # explicit index: final index is included
                             # -> 2x2

# with masking and fancy indexing
data.loc[data.popul > 15, ['area', 'density']]
```

Reading Files

2

BIS

Reading files

1. Reading Data

2. Reading CSV files and working with a dataframe

3. Categorical Variables

Reading files

```
# reading csv file
data = pd.read_csv('file_name')
# separator is ;
data = pd.read_csv('file_name', sep=';')
# decimal point is ,
data = pd.read_csv('file_name', sep='; ', decimal=',')
# if header is not in file
data = pd.read_csv('file_name', names=['n1', 'n2'])

# categorical variables, default: ordered = False
bloodtype = pd.Categorical(['O-', 'B-', 'B-', 'B-', 'A+'],
                categories=['O-','O+','B-','B+','A-','A+','AB-','AB+'])

# define columns as categorical
laptops = pd.read_csv('laptops.csv',
                      dtype={'cpu': 'category', 'brand': 'category'})
```

Operations and Missing Values in Pandas

Operating on Data in Pandas

Operations in Pandas
Ufuncs: Index Preservation
Ufuncs: Index Alignment
Operations Between DataFrame and Series

Operations on Data in Pandas

```
countries = pd.DataFrame( [ {'area': 30688}, {'area': 41850} ],
                           index=['be', 'nl'] )
population = pd.Series({'be': 11.7})
```

	area
be	30688
nl	41850

```
# create new column: index aligment
countries['pop_per_area'] = population / countries['area’]      # NaN
```

	area	pop_per_area
be	30688	0.000381
nl	41850	NaN

```
# with fill_value
countries['pop_per_area'] = population.div(countries['area'], fill_value=0)
```

	area	pop_per_area
be	30688	0.000381
nl	41850	0.000000

Python Operator	Pandas Method(s)
`+`	`add()`
`-`	`sub()`, `subtract()`
`*`	`mul()`, `multiply()`
`/`	`truediv()`, `div()`, `divide()`
`//`	`floordiv()`
`%`	`mod()`
`**`	`pow()`

Missing values

1. Handling Missing Data

2. Trade-Offs in Missing Data Conventions*

3. Missing Data in Pandas*
   - `None`: Pythonic missing data*
   - `NaN`: Missing numerical data*
   - NaN and None in Pandas*

4. Operating on Null Values
   - Detecting null values
   - Dropping null values
   - Filling null values

* Reading for context suffices

Missing Values

Pandas treats None and NaN as essentially interchangeable for indicating missing or null values

```
df = pd.DataFrame([[1,np.nan,2],
                   [2,3     ,5]])
```

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5

```
# detecting null values
df.isnull()
df.notnull()
```

	0	1	2
0	False	True	False
1	False	False	False

```
# dropping null values
df.dropna()                             # drops rows
df.dropna(axis='columns', thresh=3)  # drops columns, with min 3 Nas
```

	0	1	2
1	2.0	3.0	5

```
# filling null values
df.fillna(0)                          # fill with Nas with 0
```

4

Merge and Join

Merge and Join

1. Combining Datasets: Merge and Join

2. Relational Algebra

3. Categories of Joins
   - One-to-one joins
   - Many-to-one joins
   - Many-to-many joins

4. Specification of the Merge Key
   - The `on` keyword
   - The `left_on` and `right_on` keywords
   - The `left_index` and `right_index` keywords

5. Specifying Set Arithmetic for Joins

6. Overlapping Column Names: The `suffixes` Keyword

7. Example: US States Data

Merge and Join

```
df1 = pd.DataFrame({'employee': ['Bob', 'Jake'],
                    'group': ['Acc', 'Eng',]})
df2 = pd.DataFrame({'employee': [ 'Jake', 'Bob'],
                    'hire_date': [ 2012, 2008]})
```

df1

	employee	group
0	Bob	Acc
1	Jake	Eng

df2

	employee	hire_date
0	Jake	2012
1	Bob	2008

```
# merge detects common column
pd.merge(df1, df2)
# can merge one-to-one, many-to-one, many-to-many
```

	employee	group	hire_date
0	Bob	Acc	2008
1	Jake	Eng	2012

```
# merge with different column names
# result has both 'employee' and 'name' -> .drop('name', axis=1)
pd.merge(df1, df3, left_on="employee", right_on="name")

# merge on index, e.g. df1a = df1.set_index('employee')
df1a.join(df2a)       # pd.merge(df1a, df2a, left_index=True, right_index=True)

# merge on index and column
pd.merge(df1a, df3, left_index=True, right_on='name')
```

Merge and Join

df6

	name	food
0	Peter	fish
1	Paul	beans
2	Mary	bread

df7

	name	drink
0	Mary	wine
1	Joseph	beer

```
# default is 'inner' join
# 'outer', 'left', and 'right' joins
pd.merge(df6, df7, how='outer')
```

pd.merge(df6, df7)

	name	food	drink
0	Mary	bread	wine

pd.merge(df6, df7, how='outer')

	name	food	drink
0	Peter	fish	NaN
1	Paul	beans	NaN
2	Mary	bread	wine
3	Joseph	NaN	beer

pd.merge(df6, df7, how='left')

	name	food	drink
0	Peter	fish	NaN
1	Paul	beans	NaN
2	Mary	bread	wine

```
# overlapping column names
pd.merge(df8, df9, on="name", suffixes=["_L", "_R"])
```

df8

	name	rank
0	Bob	1
1	Jake	2
2	Lisa	3
3	Sue	4

df9

	name	rank
0	Bob	3
1	Jake	1
2	Lisa	4
3	Sue	2

pd.merge(df8, df9, on="name", suffixes=["_L", "_R"])

	name	rank_L	rank_R
0	Bob	1	3
1	Jake	2	1
2	Lisa	3	4
3	Sue	4	2

Aggregation and Grouping

Aggregation and Grouping

1. Aggregation and Grouping

2. Planets Data

3. Simple Aggregation in Pandas

4. GroupBy:
   - Split, apply, combine
   - The GroupBy object
     - Column indexing
     - Iteration over groups
     - Dispatch methods

Aggregation and Grouping

4. GroupBy: Split, Apply, Combine
   - Aggregate, filter, transform, apply
     - Aggregation
     - Filtering
     - Transformation
     - The apply() method

- Specifying the split key
    - A list, array, series, or index providing the grouping keys
    - A dictionary or series mapping index to group
    - Any Python function
    - A list of valid keys

- Grouping example

Simple Aggregation

```
df = pd.DataFrame({'A': [1, 2, 3],
                   'B': [3, 4, 5]})
```

	A	B
0	1	3
1	2	4
2	3	5

```
df.mean()   # axis=0 !
```

```
A     2.0
B     4.0
```

```
df.mean(axis=1)
```

```
0     2.0
1     3.0
2     4.0
```

```
df.describe()
```

	A	B
count	3.0	3.0
mean	2.0	4.0
std	1.0	1.0
min	1.0	3.0
25%	1.5	3.5
50%	2.0	4.0
75%	2.5	4.5
max	3.0	5.0

Aggregation	Description
`count()`	Total number of items
`first()`, `last()`	First and last item
`mean()`, `median()`	Mean and median
`min()`, `max()`	Minimum and maximum
`std()`, `var()`	Standard deviation and variance
`mad()`	Mean absolute deviation
`prod()`	Product of all items
`sum()`	Sum of all items

GroupBy

```
df = pd.DataFrame({'key':['A', 'B', 'A', 'B'],
                   'data1': [0, 1, 2, 3],
                   'data2': [2, 3, 4, 5]})
```

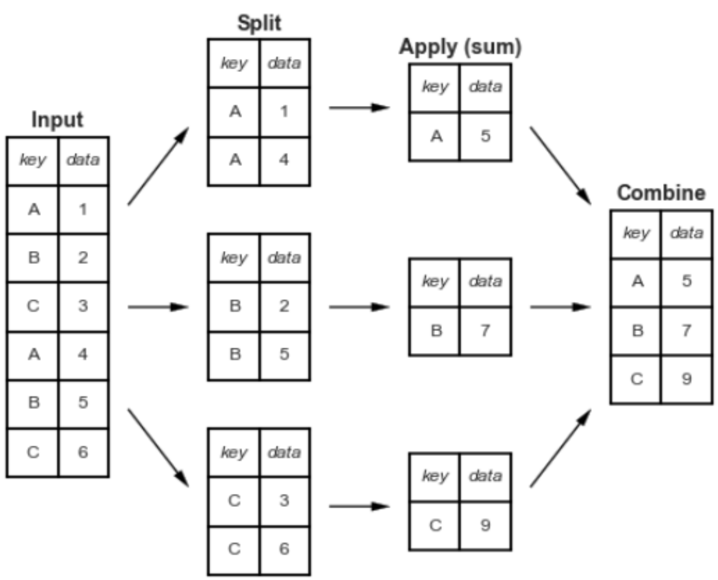
	key	data1	data2
0	A	0	2
1	B	1	3
2	A	2	4
3	B	3	5

```
# group by key and take sum
df.groupby('key').sum()
```

	data1	data2
key		
A	2	6
B	4	8

```
# grouped sum of 1 column
df.groupby('key')['data1'].sum()   # df.groupby('key').sum()['data1']
                                   # df[['key','data1']].groupby('key').sum()
```

key	
A	2
B	4



GroupBy: aggregate, apply, filter

	key	data1	data2
0	A	0	2
1	B	1	3
2	A	2	4
3	B	3	5

```
# different aggregations at ones
df.groupby('key').aggregate(['min', 'max’])
```

	data1		data2	
	min	**max**	**min**	**max**
key				
A	0	2	2	4
B	1	3	3	5

```
# different aggregations on different columns
df.groupby('key').aggregate({'data1':'min','data2':'max’})
```

```
# aggregation with own function
def norm_by_sum(x):
    return x /= x.sum()
df.groupby('key').apply(norm_by_sum)
```

key		data1	data2
A	0	0.000	0.333
A	2	1.000	0.667
B	1	0.250	0.375
B	3	0.750	0.625

```
# original df elements with grouped filter
def filter_func(x):
    return x['data2'].sum() > 6
df.groupby('key').filter(filter_func)
```

	key	data1	data2
1	B	1	3
3	B	3	5

6

Working with Strings

Working with strings

1. Vectorized String Operations

2. Introducing Pandas String Operations

3. Tables of Pandas String Methods
   - Methods similar to Python string methods
   - Methods using regular expressions
   - Miscellaneous methods
     - Vectorized item access and slicing
     - Indicator variables

Working with strings

Pandas string methods: `.str[:5], .str.len(), .str.lower(),`
`.str.contains('^a', regex=True) …`

`len()`	`lower()`	`translate()`	`islower()`
`ljust()`	`upper()`	`startswith()`	`isupper()`
`rjust()`	`find()`	`endswith()`	`isnumeric()`
`center()`	`rfind()`	`isalnum()`	`isdecimal()`
`zfill()`	`index()`	`isalpha()`	`split()`
`strip()`	`rindex()`	`isdigit()`	`rsplit()`
`rstrip()`	`capitalize()`	`isspace()`	`partition()`
`lstrip()`	`swapcase()`	`istitle()`	`rpartition()`