LEARN CODING

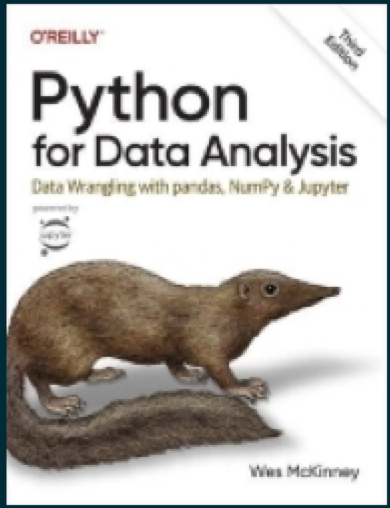
ale66

github.com/ale66/learn-coding

PANDAS

- Created by Wes McKinney, a ‘quant’ for hedge-fund AQR.
- a library for processing tabular data, both numeric and time series.
- it provides data structures (series, dataframe) and methods for data analysis.

W. McKinney, **Python for Data Analysis**, 3/e. O’Reilly 2022.




```
pip install pandas
```

DATA STRUCTURES - SERIES

SERIES

A one-dimensional object containing values and associated labels, called Index.

Unless we assign indices, Pandas will simply enumerate the items.

```python
import numpy as np
import pandas as pd
```

```python
# a simple series
s1 = pd.Series([10, 20, 30, 40])

s1
```

```
0    10
1    20
2    30
3    40
dtype: int64
```

```python
# Assign explicit indices to our data
s2 = pd.Series([10, 20, 30, 40], index = ['a', 'b', 'c', 'd'])

s2
```

```
a    10
b    20
c    30
d    40
dtype: int64
```

Example: putting 10 quid a month into a savings account

```python
my_savings = pd.Series([10, 20, 30, 40], index = ['jan', 'feb', 'mar', 'apr

my_savings
```

```
jan    10
feb    20
mar    30
apr    40
dtype: int64
```

From dictionaries to Pandas series

```
# keys correspond to indices.
my_dict = {'a':10, 'b':20, 'c':30, 'd':40}

s3 = pd.Series(my_dict)

s3
```

```
a    10
b    20
c    30
d    40
dtype: int64
```

Use the index to select one or more specific values.

```python
# Get the data on position 'a' of s3

s3['a']
```

```
np.int64(10)
```

```python
# Get the data indexed 'a' and 'c' of s3

s3[['a', 'c']]
```

```
a    10
c    30
dtype: int64
```

Filter elements

```python
# Select data which is less than 25

s3[s3<25]
```

```
a    10
b    20
dtype: int64
```

apply element-wise mathematical operations...

```
# Square every element of s3

s3**2
```

```
a     100
b     400
c     900
d    1600
dtype: int64
```

or a combination of both:

```
# Square every element of s3 smaller than 25

s3[s3<25]**2
```

```
a    100
b    400
dtype: int64
```

DATA STRUCTURES - DATAFRAMES

DATAFRAMES

2D structures where values are labelled by their index and column location.

```
# Notice how we specify columns.
new_df = pd.DataFrame([10, 20, 30, 40],
                      columns = ['Integers'],
                      index = ['a', 'b', 'c', 'd'])
new_df
```

	Integers
a	10
b	20
c	30
d	40

```python
# Implicitly add a column.
new_df['Floats'] = (1.5, 2.5, 3.5, 4.5)

new_df
```

	Integers	Floats
a	10	1.5
b	20	2.5
c	30	3.5
d	40	4.5

DATA STRUCTURES: DATAFRAME - loc

Select data according to their location label.

```
# here loc slices data using index name.

new_df.loc['c']
```

```
Integers    30.0
Floats       3.5
Name: c, dtype: float64
```

```
# here loc slices data using column name.

new_df.loc[:, 'Integers'] #or new_df['numbers']
```

```
a    10
b    20
c    30
d    40
Name: Integers, dtype: int64
```

```
# here we use both index and column name.

new_df.loc['c', 'Integers']
```

```
np.int64(30)
```

DATA STRUCTURES: DATAFRAME - `iloc`

Select a specific slice of data according to its position.

```python
# here loc slices data using index number.
new_df.iloc[2]
```

```
Integers    30.0
Floats       3.5
Name: c, dtype: float64
```

```python
# here loc slices data using column number.
new_df.iloc[:, 0]
```

```
a    10
b    20
c    30
d    40
Name: Integers, dtype: int64
```

```python
# here we use both index and column number.
new_df.iloc[2, 0]
```

```
np.int64(30)
```

DATA STRUCTURES: DATAFRAME - FILTERS

Complex selection is achieved applying Boolean filters.
Multiple conditions can be combined in one statement.

```
new_df[new_df['Integers']>10]
```

	Integers	Floats
b	20	2.5
c	30	3.5
d	40	4.5

```
# here we apply conditions to both columns.

new_df[(new_df.Integers>10) & (new_df.Floats>2.5)]
```

	Integers	Floats
c	30	3.5
d	40	4.5

DATA STRUCTURES: DATAFRAME - `Axis`

DataFrames operate on 2 dimensions.

`Axis = 0` invokes functions across rows

default behaviour when the axis is not specified.

```
1  new_df.sum()
```

```
Integers    100.0
Floats       12.0
dtype: float64
```

`Axis = 1` invokes functions across columns.

```
1  new_df.sum(axis=1)
```

```
a    11.5
b    22.5
c    33.5
d    44.5
dtype: float64
```

We can mix element-wise operations with axis functions

Example: Create a column with the sum of squares of each row.

```
# Just one line of code!
new_df['Sumsq'] = (new_df**2).sum(axis=1)
new_df
```

	Integers	Floats	Sumsq
a	10	1.5	102.25
b	20	2.5	406.25
c	30	3.5	912.25
d	40	4.5	1620.25

FROM NUMPY TO PANDAS

FROM NUMPY TO PANDAS: `where()`

Pandas executes `where()` differently

Numpy allows specifying the respective action associated to `True` and `False`

```
l = np.arange(9).reshape((3, 3))
l
```

```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

```
#  If True then make it double, else halve it
np.where(l<5, l*2, l/2)
```

```
array([[0. , 2. , 4. ],
       [6. , 8. , 2.5],
       [3. , 3.5, 4. ]])
```

Pandas assigns `n/a` when `False`

NUMPY FUNC. TO PANDAS OBJECTS

```python
# l is a Numpy matrix which readily interoperates with Pandas
my_df = pd.DataFrame(l, columns=['A', 'B', 'C'])

my_df
```

	A	B	C
0	0	1	2
1	3	4	5
2	6	7	8

```python
# Extract the square root of each el. of column B (NB: my_df remains unchan
np.sqrt(my_df.B)
```

```
0    1.000000
1    2.000000
2    2.645751
Name: B, dtype: float64
```

BACK AND FORTH B/W PANDAS AND NUMPY

```python
# Extract the values back into a Numpy object
m = my_df.values
m
```

```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

IMPORTING DATA

Read a datafile and turn it into a DataFrame. Several arguments are available to specify the behavior of the process:

`index_col` sets the column of the csv file to be used as index of the DataFrame

`sep` specifies the separator in the source file

`parse_dates` sets the cols. to be converted into *datetimes*

```
FILE = './path/to/some_file.csv'

df_r = pd.read_csv(FILE,
                   index_col = 0,
                   sep = ';',
                   parse_dates = ['date'] )
```

BIOSTATS DATA - `info()`

The `info()` method outputs top-down information on the DataFrame

```python
MYDATA = 'data/biostats.csv'

df_bio = pd.read_csv(MYDATA)

df_bio.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 18 entries, 0 to 17
Data columns (total 5 columns):
 #   Column        Non-Null Count  Dtype 
---  ------        --------------  ----- 
 0   Name          18 non-null     object
 1   Sex           18 non-null     object
 2   Age           18 non-null     int64 
 3   Height(in)    18 non-null     int64 
 4   Weight(lbs)   18 non-null     int64 
dtypes: int64(3), object(2)
memory usage: 852.0+ bytes
```

BIOSTATS DATA - `head()` AND `tail()`

Handy visualisation of first/last n rows (default = 5)

```
df_bio.head()
```

	Name	Sex	Age	Height(in)	Weight(lbs)
0	Alex	M	41	74	170
1	Bert	M	42	68	166
2	Dave	M	32	70	155
3	Dave	M	39	72	167
4	Elly	F	30	66	124

```
1 df_bio.tail()
```

	Name	Sex	Age	Height(in)	Weight(lbs)
13	Neil	M	36	75	160
14	Omar	M	38	70	145
15	Page	F	31	67	135
16	Luke	M	29	71	176
17	Ruth	F	28	65	131

BIOSTATS DATA - INDEX COLUMN

Selecting the index column affects the structure of the DataFrame and thus information retrieval.

Caution: the index does not have to be unique. Multiple rows could have the same index name.

```
# here we set the Name column as the index
df_bio2 = pd.read_csv(MYDATA, index_col = 0)

df_bio2.head(2)
```

	Sex	Age	Height(in)	Weight(lbs)
Name				
Alex	M	41	74	170
Bert	M	42	68	166

```python
#It is now possible to use elements of the Name column to select an entire
df_bio2.loc['Bert']
```

```
Sex                M
Age               42
Height(in)        68
Weight(lbs)      166
Name: Bert, dtype: object
```

DESCRIPTIVE STATS - describe()

```python
# Compute the descriptive stats of quantitative variables
df_bio.describe()
```

	Age	Height(in)	Weight(lbs)
count	18.000000	18.000000	18.000000
mean	34.666667	69.055556	146.722222
std	7.577055	3.522570	22.540958
min	23.000000	62.000000	98.000000
25%	30.000000	66.250000	132.000000
50%	32.500000	69.500000	150.000000
75%	38.750000	71.750000	165.250000

	Age	Height(in)	Weight(lbs)
max	53.000000	75.000000	176.000000

Also on selections:

```python
# Descriptive statistics for the Age variable
df_bio['Age'].describe()
```

```
count    18.000000
mean     34.666667
std       7.577055
min      23.000000
25%      30.000000
50%      32.500000
75%      38.750000
max      53.000000
Name: Age, dtype: float64
```

DESCRIPTIVE STATS - CATEGORICAL VARS

The `value_counts()` method computes the unique values and how many times they occur.

```
# Descriptive statistics for the entire DataFrame
df_bio.Sex.value_counts()
```

```
Sex
M     11
F      7
Name: count, dtype: int64
```

PANDAS DATA DISPLAYS

Pandas objects come with methods for visualisation

they are built on top of matplotlib

```
df_bio['Age'].plot(kind = 'hist')

# alternative syntax:
# df_bio.Age.plot(kind = 'hist')
```