pandas

Introduction to pandas data structures

What is pandas

pandas is an open source Python library for data analysis. Python has always been great for prepping and munging data, but it's never been great for analysis - you'd usually end up using R or loading it into a database and using SQL (or worse, Excel). pandas makes Python great for analysis.

Data Structures

pandas introduces two new data structures to Python which are built on top of NumPy (this means it's fast):

- Series
- DataFrame

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
pd.set_option('max_columns', 50)
%matplotlib inline
```

Series

A Series is a one-dimensional object similar to an array, list, or column in a table. It will assign a labeled **index** to each item in the Series. By default, each item will receive an index label from 0 to N, where N is the length of the Series minus one.

```
# create a Series with an arbitrary list
s = pd.Series([7, 'Heisenberg', 3.14, -1789710578, 'Happy Eating!'])
s
```

```
0                 7
1       Heisenberg
2              3.14
3      -1789710578
4     Happy Eating!
dtype: object
```

Series with index

you can specify an index to use when creating the Series

```
s = pd.Series([7, 'Heisenberg', 3.14, -1789710578, 'Happy Eating!'],
              index=['A', 'Z', 'C', 'Y', 'E'])
s
A                  7
Z       Heisenberg
C             3.14
Y      -1789710578
E    Happy Eating!
dtype: object
```

Dictionary to Series

The Series constructor can convert a dictionary as well, using the keys of the dictionary as its index.

```
d = {'Chicago': 1000, 'New York': 1300, 'Portland': 900, 'San
Francisco': 1100,
     'Austin': 450, 'Boston': None}
cities = pd.Series(d)
cities
```

```
Austin            450
Boston            NaN
Chicago          1000
New York         1300
Portland          900
San Francisco    1100
dtype: float64
```

Access Series' elements by index

You can use the index to select specific items from the Series ...

```
cities['Chicago']
1000.0

cities[['Chicago', 'Portland', 'San Francisco']]
Chicago          1000
Portland          900
San Francisco    1100
dtype: float64
```

Boolean indexing

You can use boolean indexing for selection.

```
cities[cities < 1000]
Austin      450
Portland    900
dtype: float64
```

cities < 1000 returns a Series of True/False values, which we then pass to our Series cities, returning the corresponding True items.

```
less_than_1000 = cities < 1000
less_than_1000
Austin           True
Boston          False
Chicago         False
New York        False
Portland         True
San Francisco   False
dtype: bool
```

```
cities[less_than_1000]
Austin      450
Portland    900
dtype: float64
```

Editing Series' elements

You can also change the values in a Series on the fly

```
# changing based on the index
print('Old value:', cities['Chicago'])
cities['Chicago'] = 1400
print('New value:', cities['Chicago'])
('Old value:', 1000.0)
('New value:', 1400.0)
```

```
# changing values using
# boolean logic
cities[cities < 1000]
Austin      450
Portland    900
dtype: float64

cities[cities < 1000] = 750
cities[cities < 1000]
Austin      750
Portland    750
dtype: float64
```

Element membership

You can check whether an element belongs to a Series using idiomatic Python

```
print('Seattle' in cities)
print('San Francisco' in cities)
```

```
False
True
```

Mathematical operations on Series

Mathematical operations can be done using scalars and functions.

```
# divide city values by 3
cities / 3
```

```
Austin           250.000000
Boston                  NaN
Chicago          466.666667
New York         433.333333
Portland         250.000000
San Francisco    366.666667
dtype: float64
```

```
# square city values
np.square(cities)
```

```
Austin           562500
Boston              NaN
Chicago          1960000
New York         1690000
Portland          562500
San Francisco    1210000
dtype: float64
```

Adding Series

Adding two Series together, which returns a union of the two Series with the addition occurring on the shared index values. Values on either Series that did not have a shared index will produce a NULL/NaN (not a number)

```
cities[['Chicago', 'New York']]
Chicago     1400
New York    1300
dtype: float64

cities[['Austin', 'New York']]
Austin       750
New York    1300
dtype: float64

cities[['Chicago', 'New York']] + cities[['Austin', 'New York']]
Austin       NaN
Chicago      NaN
New York    2600
```

DataFrame

A DataFrame is a tabular data structure comprised of rows and columns, akin to a spreadsheet, database table, or R's data.frame object. You can also think of a DataFrame as a group of Series objects that share an index (the column names).

Reading data into DataFrame

To create a DataFrame pass a dictionary of lists to the DataFrame constructor

```
data = {'year': [2010, 2011, 2012, 2011, 2012, 2010, 2011, 2012],
        'team': ['Bears', 'Bears', 'Bears', 'Packers', 'Packers', 'Lions', 'Lions',
'Lions'],
        'wins': [11, 8, 10, 15, 11, 6, 10, 4],
        'losses': [5, 8, 6, 1, 5, 10, 6, 12]}
football = pd.DataFrame(data, columns=['year', 'team', 'wins', 'losses'])
football
```

	year	team	wins	losses
0	2010	Bears	11	5
1	2011	Bears	8	8
2	2012	Bears	10	6
3	2011	Packers	15	1
4	2012	Packers	11	5
5	2010	Lions	6	10
6	2011	Lions	10	6
7	2012	Lions	4	12

CSV -> DataFrame

Reading a CSV is as simple as calling the **`read_csv`** function. By default, the **`read_csv`** function expects the column separator to be a comma, but you can change that using the **`sep`** parameter.

```
%cd ~/path_to_csv_dir/
/Users/sergei/path_to_csv_dir

!head -n 3 fugitives.csv
Fugitive,Nationality,Wanted by,Wanted for,Details of reason wanted for,
Viktoryia TSUNIK,Belarus,Belarus,"Theft, Fraud",Theft by abuse of power
Adriano GIACOBONE,Italy,Italy,"Kidnapping, Possession of firearms and/or
Sudiman SUNOTO,Indonesia,Indonesia,"Illegal Logging, Environmental Crimes",

from_csv = pd.read_csv('fugitives.csv')
from_csv.head()
```

CSV -> DataFrame (cont.)

Specify separator and which columns to read

```
cols = ['Fugitive', 'Nationality', 'Wanted by']
no_headers = pd.read_csv('fugitives.csv', sep=‘,',
                         usecols=cols)
no_headers.head()
```

DataFrame -> CSV

Save DataFrame to CSV fike

```
my_dataframe.to_csv('path_to_file.csv')
```

See http://pandas.pydata.org/pandas-docs/stable/io.html for more details on pandas Input/Output

Database -> DataFrame

pandas also has some support for reading/writing DataFrames directly from/to a database.

You'll typically just need to pass a *connection object* or *sqlalchemy* engine to the **`read_sql`** or **`to_sql`** functions within the **`pandas.io`** module.

```
from pandas.io import sql
import psycopg2

conn = conn = psycopg2.connect('postgres://user:password@host/db')
query = "SELECT * FROM towed WHERE make = 'FORD';"

results = sql.read_sql(query, con=conn)
results.head()
```

Clipboard -> DataFrame

Write delimited data you've copied to your clipboard into a DataFrame.

```
foo = pd.read_clipboard()
foo.head()
```

The function does a good job of inferring the delimiter, but you can also use the sep parameter to be explicit.

URL -> DataFrame

With read_table, we can also read directly from a URL pointing to a delimited data

```
url = 'https://raw.githubusercontent.com/evdoks/data_science/' \
      'master/data/fugitives.csv'

# fetch the text from the URL and read it into a DataFrame
from_url = pd.read_table(url, sep=',')
from_url.head(3)
```

pandas

Working with DataFrames

MovieLens Dataset

MovieLens dataset contains 100,000 ratings made by 943 users on 1,682 movies.

https://grouplens.org/datasets/movielens/

```
# pass in column names for each CSV
u_cols = ['user_id', 'age', 'sex', 'occupation', 'zip_code']
users = pd.read_csv('https://raw.githubusercontent.com/evdoks/data_science/'
                    'master/data/ml-100k/u.user', sep='|', names=u_cols,
                    encoding='latin-1')

r_cols = ['user_id', 'movie_id', 'rating', 'unix_timestamp']
ratings = pd.read_csv('https://raw.githubusercontent.com/evdoks/data_science/'
                      'master/data/ml-100k/u.data', sep='\t', names=r_cols,
                      encoding='latin-1')

# the movies file contains columns indicating the movie's genres
# let's only load the first five columns of the file with 'usecols'
m_cols = ['movie_id', 'title', 'release_date', 'video_release_date', 'imdb_url']
movies = pd.read_csv('https://raw.githubusercontent.com/evdoks/data_science/'
                     'master/data/ml-100k/u.item', sep='|', names=m_cols,
                     usecols=range(5), encoding='latin-1')
```

Inspection

pandas has a variety of functions for getting basic information about your DataFrame, the most basic of which is using the `info` method.

```
movies.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1682 entries, 0 to 1681
Data columns (total 5 columns):
movie_id                1682 non-null int64
title                   1682 non-null object
release_date            1681 non-null object
video_release_date      0 non-null float64
imdb_url                1679 non-null object
dtypes: float64(1), int64(1), object(3)
memory usage: 78.8+ KB
```

Inspection (cont.)

Information about attributes of a DataFrame and their data types

```
movies.dtypes
movie_id                int64
title                  object
release_date           object
video_release_date    float64
imdb_url               object
dtype: object
```

Inspection (cont.)

DataFrame's also have a describe method, which is outputs basic statistics about the dataset's numeric columns.

```
users.describe()
```

	user_id	age
count	943.000000	943.000000
mean	472.000000	34.051962
std	272.364951	12.192740
min	1.000000	7.000000
25 %	236.500000	25.000000
50 %	472.000000	31.000000
75 %	707.500000	43.000000
max	943.000000	73.000000

Outputting DataFrame

`head` displays the first five records of the dataset

```
movies.head()
```

`tail` displays the first five records of the dataset

```
movies.tail()
```

Python's regular slicing syntax works as well

```
movies[20:22]
```

Selecting columns

Selecting a single column from the DataFrame will return a Series object.

```
users['occupation'].head()
```

```
0    technician
1         other
2        writer
3    technician
4         other
Name: occupation, dtype: object
```

Selecting multiple columns

To select multiple columns, simply pass a list of column names to the DataFrame, the output of which will be a DataFrame.

```
users[['age', ‘zip_code']]
      .head()
```

```
   age zip_code
0   24    85711
1   53    94043
2   23    32067
3   24    43537
4   33    15213
```

```
# can also store in a variable
columns_you_want = ['occupation', 'sex']
users[columns_you_want].head()
```

```
   occupation sex
0  technician   M
1       other   F
2      writer   M
3  technician   M
4       other   F
```

Selecting rows

Selection by an individual index or boolean indexing

```
# users older than 25
print(users[users.age > 25].head(3))
print('\n')

# users aged 40 AND male
print(users[(users.age == 40) & (users.sex == 'M')].head(3))
print('\n')

# users younger than 30 OR female
print(users[(users.sex == 'F') | (users.age < 30)].head(3))
```

Reindexing

`set_index` returns a new DataFrame with a new index

```
users.set_index('user_id').head()

# the DataFrame was not changed
users.head()

# set_index actually returns a new DataFrame
with_new_index = users.set_index('user_id')
with_new_index.head()
```

Use **`inplace`** to modify an existing DataFrame

```
users.set_index('user_id', inplace=True)
users.head()
```

Selecting rows by position and index label

Rows can be selected by position using the **`iloc`** method

```
users.iloc[99]
users.iloc[[1, 50, 300]]
```

Rows can be selected by index label using the **`loc`** method

```
users.loc[100]
users.loc[[2, 51, 301]]
```

Resetting index

It is possible to reset index

```
users.reset_index(inplace=True)
users.head()
```

	user_id	age	sex	occupation	zip_code
0	1	24	M	technician	85711
1	2	53	F	other	94043
2	3	23	M	writer	32067
3	4	24	M	technician	43537
4	5	33	F	other	15213

pandas offers numerous other ways to do selection:

http://pandas.pydata.org/pandas-docs/stable/indexing.html

Joining

A SQL *join* clause combines columns from one or more tables in a relational database.

We will consider following types of a *join:*

- inner join
- left join
- right join
- outer join

Sample data

Employee table

LastName	DepartmentID
Rafferty	31
Jones	33
Heisenberg	33
Robinson	34
Smith	34
Williams	NULL

Department table

DepartmentID	DepartmentName
31	Sales
33	Engineering
34	Clerical
35	Marketing

Inner join

- An *inner join* requires each row in the two joined tables to have matching column values, and is a commonly used join operation
- Inner join creates a new result table by combining column values of two tables (A and B) based upon the *join-condition*
- The query compares each row of A with each row of B to find all pairs of rows which satisfy the join-condition. When the join-condition is satisfied by matching non-NULL values, column values for each matched pair of rows of A and B are combined into a result row.

Inner join example

```
SELECT employee.LastName, employee.DepartmentID,
department.DepartmentName
FROM employee
INNER JOIN department ON
employee.DepartmentID = department.DepartmentID
```

Employee.LastName	Employee.DepartmentID	Department.DepartmentName
Robinson	34	Clerical
Jones	33	Engineering
Smith	34	Clerical
Heisenberg	33	Engineering
Rafferty	31	Sales

Outer join

The outer joined table retains each row—even if no other matching row exists.

There three different outer join types:

- left outer join
- right outer join
- full outer join

Left outer join

Left outer join returns all the values from an inner join plus all values in the left table that do not match to the right table, including rows with NULL (empty) values in the link column.

```
SELECT *
FROM employee e
LEFT OUTER JOIN department d ON e.DepartmentID = d.DepartmentID;
```

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID
Jones	33	Engineering	33
Rafferty	31	Sales	31
Robinson	34	Clerical	34
Smith	34	Clerical	34
Williams	NULL	NULL	NULL
Heisenberg	33	Engineering	33

Right outer join

Right outer join returns all the values from an inner join plus all values in the right table that do not match to the left table, including rows with NULL (empty) values in the right column.

```
SELECT *
FROM employee e RIGHT OUTER JOIN department d
  ON e.DepartmentID = d.DepartmentID;
```

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID
Smith	34	Clerical	34
Jones	33	Engineering	33
Robinson	34	Clerical	34
Heisenberg	33	Engineering	33
Rafferty	31	Sales	31
NULL	NULL	*Marketing*	*35*

Full outer join

Full outer join combines the effect of applying both left and right outer joins. Where rows in the FULL OUTER JOINed tables do not match, the result set will have NULL values for every column of the table that lacks a matching row

```
SELECT *
FROM employee e FULL OUTER JOIN department d
  ON e.DepartmentID = d.DepartmentID;
```

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID
Smith	34	Clerical	34
Jones	33	Engineering	33
Robinson	34	Clerical	34
Williams	NULL	NULL	NULL
Heisenberg	33	Engineering	33
Rafferty	31	Sales	31
NULL	NULL	*Marketing*	*35*

pandas merge

- **`pandas.merge`** allows two DataFrames to be joined on one or more keys
- the function provides a series of parameters (**`on`**, **`left_on`**, **`right_on`**, **`left_index`**, **`right_index`**) allowing you to specify the columns or indexes on which to join
- by default, **`pandas.merge`** operates as an inner join

Sample data

```
df_employee = pd.DataFrame(
{'LastName': ['Rafferty', 'Jones', 'Heisenberg', 'Robinson',
       'Smith', 'Williams'],
'DepartmentId': [31, 33, 33, 34, 34, np.nan]
})


df_department = pd.DataFrame(
  {'DepartmentName': ['Sales', 'Engineering', 'Clerical',
   'Marketing']},
  index = [31, 33, 34, 35]
)
```

pandas inner join

```
pd.merge(df_employee, df_department, left_on='DepartmentId',
         right_index=True, how='inner')
```

	DepartmentId	LastName	DepartmentName
0	31.0	Rafferty	Sales
1	33.0	Jones	Engineering
2	33.0	Heisenberg	Engineering
3	34.0	Robinson	Clerical
4	34.0	Smith	Clerical

pandas left outer join

```
pd.merge(df_employee, df_department, left_on='DepartmentId',
         right_index=True, how='left')
```

	DepartmentId	LastName	DepartmentName
0	31.0	Rafferty	Sales
1	33.0	Jones	Engineering
2	33.0	Heisenberg	Engineering
3	34.0	Robinson	Clerical
4	34.0	Smith	Clerical
5	NaN	Williams	NaN

right outer join

```
pd.merge(df_employee, df_department, left_on='DepartmentId',
         right_index=True, how='right')
```

	DepartmentId	LastName	DepartmentName
0	31.0	Rafferty	Sales
1	33.0	Jones	Engineering
2	33.0	Heisenberg	Engineering
3	34.0	Robinson	Clerical
4	34.0	Smith	Clerical
5	35.0	NaN	Marketing

full outer join

```
pd.merge(df_employee, df_department, left_on='DepartmentId',
         right_index=True, how='outer')
```

	DepartmentId	LastName	DepartmentName
0	31.0	Rafferty	Sales
1	33.0	Jones	Engineering
2	33.0	Heisenberg	Engineering
3	34.0	Robinson	Clerical
4	34.0	Smith	Clerical
5	NaN	Williams	NaN
5	35.0	NaN	Marketing

Combining DataFrames

pandas also provides a way to combine DataFrames along an axis - **`pandas.concat`**

```
df_employee_1 = pd.DataFrame(
  {'LastName': ['Guenther', 'Schulz'],
  'DepartmentId': [31, 33]}
  )
```

```
pd.concat([df_employee, df_employee_1])
```

	DepartmentId	LastName
0	31.0	Rafferty
1	33.0	Jones
2	33.0	Heisenberg
3	34.0	Robinson
4	34.0	Smith
5	NaN	Williams
0	31.0	Guenther
1	33.0	Schulz

Combining DataFrames (cont.)

By default, the function will vertically append the objects to one another, combining columns with the same name. We can see above that values not matching up will be NULL.

```
pd.concat([df_employee, df_department])
```

	DepartmentId	DepartmentName	LastName
0	31.0	NaN	Rafferty
1	33.0	NaN	Jones
2	33.0	NaN	Heisenberg
3	34.0	NaN	Robinson
4	34.0	NaN	Smith
5	NaN	NaN	Williams
31	NaN	Sales	NaN
33	NaN	Engineering	NaN
34	NaN	Clerical	NaN
35	NaN	Marketing	NaN

Grouping and Aggregating DataFrames

City of Chicago dataset

We will use publicly available data about salaries of city of Chicago employees

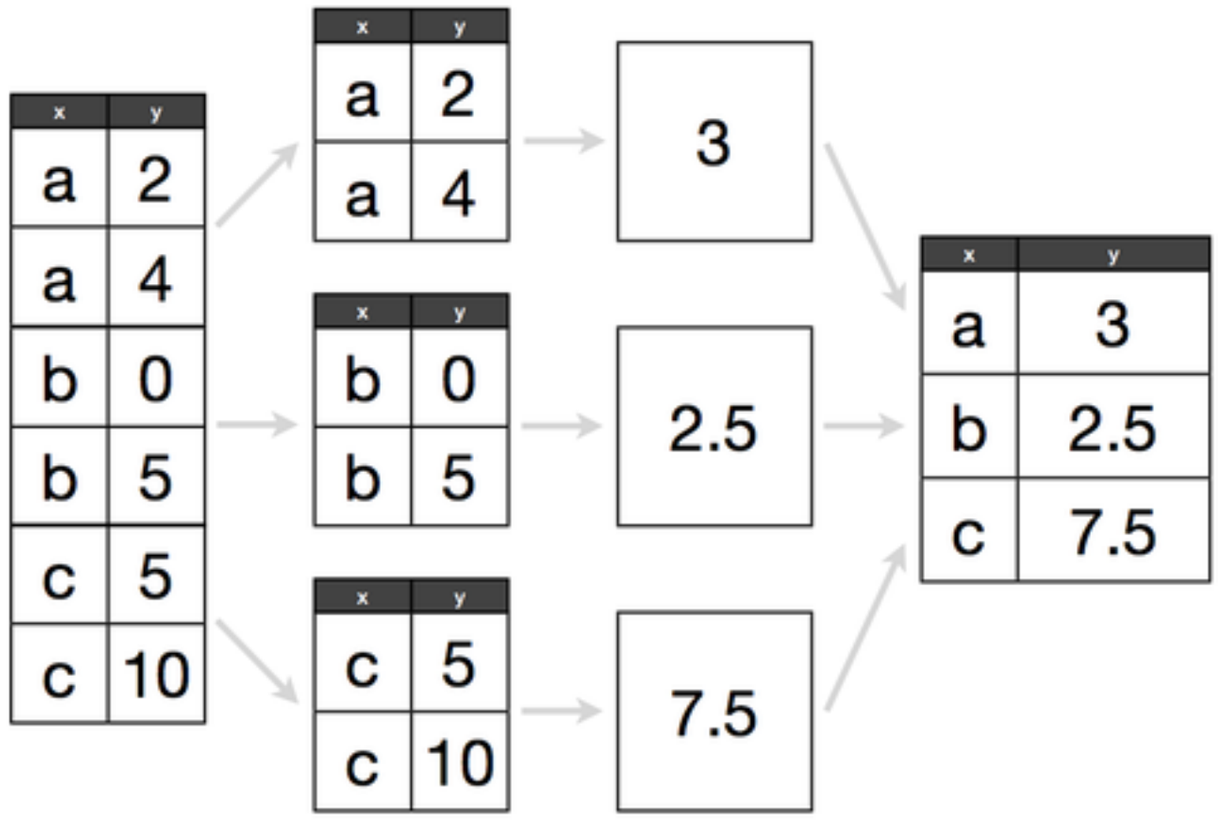
```
url = 'https://raw.githubusercontent.com/evdoks/data_science/' \
      'master/data/city-of-chicago-salaries.csv'
headers = ['name', 'title', 'department', 'salary']
chicago = pd.read_csv(url, sep=',', header=0, names=headers,
                      converters = {
                          'salary':
                          lambda x: float(x.replace('$', ''))
                      }
                     )
chicago.head()
```

City of Chicago dataset (cont.)

	name	title	department	salary
0	AARON, ELVIA	WATER RATE	WATER MGMNT	85512
1	AARON,	POLICE OFFICER	POLICE	75372
2	AARON,	CHIEF CONTRACT	GENERAL	80916
3	ABAD JR,	CIVIL ENGINEER	WATER MGMNT	99648
4	ABBATACOLA,	ELECTRICAL	AVIATION	89440

Grouping

Assume we have a DataFrame and want to get the average for each group - visually, the split-apply-combine method looks like this:



pandas groupby

pandas **`groupby`** returns a **`DataFrameGroupBy`** object which has a variety of methods, many of which are similar to standard SQL aggregate functions.

```
by_dept = chicago.groupby('department')
by_dept
<pandas.core.groupby.DataFrameGroupBy object at 0x1128ca1d0>
```

By default, **`groupby`** turns the grouped field into an index

Aggregation functions: count

Calling **`count`** returns the total number of NOT NULL values within each column.

```
by_dept.count().head() # NOT NULL records within each column
```

```
                      name  title  salary
department
ADMIN HEARNG            42     42      42
ANIMAL CONTRL           61     61      61
AVIATION              1218   1218    1218
BOARD OF ELECTION      110    110     110
BOARD OF ETHICS          9      9       9
```

```
SELECT COUNT(*) FROM chicago c
WHERE c.name IS NOTNOT NULL
GROUP BY c.department
```

Aggregation functions: size

Calling **`size`** returns the total number of records in each group.

```
by_dept.size().head() # total records for each department
```

```
department
PUBLIC LIBRARY     926
STREETS & SAN     2070
TRANSPORTN        1168
TREASURER           25
WATER MGMNT       1857
dtype: int64
```

```
SELECT COUNT(*) FROM chicago c
GROUP BY c.department
```

Aggregation functions: sum

Calling **`sum`** returns the sum of numerical records in each group.

```
by_dept.sum()[20:25] # total salaries of each department
# by_dept['salary'].sum()[20:25].head()
```

```
                          salary
department
HUMAN RESOURCES      4850928.0
INSPECTOR GEN        4035150.0
IPRA                 7006128.0
LAW                 31883920.2
LICENSE APPL COMM      65436.0
```

```
SELECT SUM(c.salary) FROM chicago c
GROUP BY c.department
```

Aggregation functions: mean

Calling **`mean`** returns the mean of numerical records in each group.

```
by_dept.mean()[20:25] # average salary of each department
```

```
                          salary
department
HUMAN RESOURCES     71337.176471
INSPECTOR GEN       80703.000000
IPRA                82425.035294
LAW                 70853.156000
LICENSE APPL COMM   65436.000000
```

```
SELECT MEAN(c.salary) FROM chicago c
GROUP BY c.department
```

Aggregation functions: median

Calling **`median`** returns the mean of numerical records in each group.

```
by_dept.median()[20:25]  # median salary of each department
```

```
                        salary
department
HUMAN RESOURCES          68496
INSPECTOR GEN            76116
IPRA                     82524
LAW                      66492
LICENSE APPL COMM        65436
```

```
SELECT … ???
```

Computing median is not available in most database management systems.