2. Hello, pandas!

February 6, 2021

1 2. Hello, pandas!

This notebook will introduce you to the `pandas` data analysis library and demonstrate how to load, inspect, sort, filter, group and aggregate a data set.

The data for this exercise will be a CSV of USA TODAY's opening-day MLB salaries from the 2018 season.

Session outline

- Import pandas
- Load data into a data frame
- Inspect the data
- Sort the data
- Filter the data
- Group and aggregate the data
- Export to CSV

1.0.1 Import pandas

Before you can use the functionality of `pandas`, a third-party library installed separately from Python, you need to *import* it. The convention is to import the library under an alias that's easier to type: `as pd`.

Run this cell:

[44]:
```
import pandas as pd
```

1.0.2 Change a display setting

Run the next cell to change a setting that displays big numbers in scientific notation by default. (Unless scientific notation is your jam, in which case *avoid* running the next cell.)

[45]:
```
# found via googling!
# https://pandas.pydata.org/pandas-docs/stable/user_guide/options.html
pd.options.display.float_format = '{:20,.2f}'.format
```

1.0.3 Load data into a data frame

Before you can start poking at a data file, you need to load the data into a pandas *data frame*, which is sort of like a virtual spreadsheet with columns and rows.

You can load many different types of data files into a data frame, including CSVs (and other delimited text files), Excel files, JSON and more. (Here's a quick reference notebook demonstrating how to import some different data files, including live data from the Internet!)

For today, we'll focus on importing the MLB salary data using a pandas method called `read_csv()`. There are a ton of options you can supply when you read in the data file, but at minimum, you need to tell the method *where* the file lives, which means you need to supply the path to the data file as a Python *string* (some text enclosed in single or double quotes). The file is called `mlb.csv`, and it is located in the same directory as this notebook file, so we don't need to specify a longer path.

As we import the data, we'll also *assign* the results of the loading operation to a new variable called *df* (short for data frame – easy to type, plus you'll see this pattern a lot when Googling around for help).

[46]:
```
df = pd.read_csv('../data/mlb.csv')
```

As a human sentence: "Go to the pandas library that we imported earlier as something called `pd` and use its `read_csv()` method to import a file called `mlb.csv` into a data frame – and while we're at it, assign the results of that operation to a new variable called `df`."

1.0.4 Inspect the data

Let's take a look at what we've got using a few built-in methods and attributes of a pandas data frame: - `df.info()` will display basic information about your dataframe - `df.head()` will display the first five records (or, if you prefer, you can specify a number, e.g., `df.head(10)`) - `df.tail()` will display the last five records (or, if you prefer, you can specify a number, e.g., `df.tail(10)`) - `df.describe()` will compute summary stats on numeric columns - `df.sample()` will return a randomly selected record (or, if you prefer, you specify a number, e.g., `df.sample(5)` - `df.columns` will list column names (you can use the `rename()` method to rename them - `df.shape` will tell you how many columns, how many rows - `df.dtypes` will list the column names and tell you what kind of data is in each one

[47]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 868 entries, 0 to 867
Data columns (total 7 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   NAME        868 non-null    object
 1   TEAM        868 non-null    object
 2   POS         868 non-null    object
 3   SALARY      868 non-null    int64
```

```
 4   START_YEAR  868 non-null    int64
 5   END_YEAR    868 non-null    int64
 6   YEARS       868 non-null    int64
dtypes: int64(4), object(3)
memory usage: 47.6+ KB
```

[48]:
```
df.head()
```

[48]:

	NAME	TEAM	POS	SALARY	START_YEAR	END_YEAR	YEARS
0	Clayton Kershaw	LAD	SP	33000000	2014	2020	7
1	Zack Greinke	ARI	SP	31876966	2016	2021	6
2	David Price	BOS	SP	30000000	2016	2022	7
3	Miguel Cabrera	DET	1B	28000000	2014	2023	10
4	Justin Verlander	DET	SP	28000000	2013	2019	7

[49]:
```
df.head(10)
```

[49]:

	NAME	TEAM	POS	SALARY	START_YEAR	END_YEAR	YEARS
0	Clayton Kershaw	LAD	SP	33000000	2014	2020	7
1	Zack Greinke	ARI	SP	31876966	2016	2021	6
2	David Price	BOS	SP	30000000	2016	2022	7
3	Miguel Cabrera	DET	1B	28000000	2014	2023	10
4	Justin Verlander	DET	SP	28000000	2013	2019	7
5	Jason Heyward	CHC	RF	26055288	2016	2023	8
6	Albert Pujols	LAA	1B	26000000	2012	2021	10
7	Felix Hernandez	SEA	SP	25857143	2013	2019	7
8	Jon Lester	CHC	SP	25000000	2015	2020	6
9	CC Sabathia	NYY	SP	25000000	2012	2016	5

[50]:
```
df.tail()
```

[50]:

	NAME	TEAM	POS	SALARY	START_YEAR	END_YEAR	YEARS
863	Steve Selsky	BOS	RF	535000	2017	2017	1
864	Stuart Turner	CIN	C	535000	2017	2017	1
865	Vicente Campos	LAA	RP	535000	2017	2017	1
866	Wandy Peralta	CIN	RP	535000	2017	2017	1
867	Yandy Diaz	CLE	3B	535000	2017	2017	1

[51]:
```
df.tail(7)
```

[51]:

	NAME	TEAM	POS	SALARY	START_YEAR	END_YEAR	YEARS
861	Rookie Davis	CIN	SP	535000	2017	2017	1
862	Stephen Cardullo	COL	1B	535000	2017	2017	1
863	Steve Selsky	BOS	RF	535000	2017	2017	1
864	Stuart Turner	CIN	C	535000	2017	2017	1
865	Vicente Campos	LAA	RP	535000	2017	2017	1
866	Wandy Peralta	CIN	RP	535000	2017	2017	1

```
867        Yandy Diaz  CLE  3B  535000        2017      2017      1
```

[52]:
```
df.describe()
```

[52]:
```
                  SALARY           START_YEAR             END_YEAR  \
count              868.00               868.00               868.00
mean         4,468,069.18             2,016.49             2,017.43
std          5,948,459.31                 1.21                 1.16
min            535,000.00             2,008.00             2,015.00
25%            545,500.00             2,017.00             2,017.00
50%          1,562,500.00             2,017.00             2,017.00
75%          6,000,000.00             2,017.00             2,017.00
max         33,000,000.00             2,017.00             2,027.00

                   YEARS
count              868.00
mean                1.94
std                 1.92
min                 1.00
25%                 1.00
50%                 1.00
75%                 2.00
max                13.00
```

[53]:
```
df.sample()
```

[53]:
```
            NAME TEAM POS  SALARY  START_YEAR  END_YEAR  YEARS
652  Matt Duffy   TB  3B  545300        2017      2017      1
```

[54]:
```
df.sample(5)
```

[54]:
```
              NAME TEAM POS    SALARY  START_YEAR  END_YEAR  YEARS
559  J.T. Realmuto  MIA   C    562500        2017      2017      1
673    Aaron Judge  NYY  RF    544500        2017      2017      1
516     Eric Fryer  STL   C    675000        2017      2017      1
121    Ian Kinsler  DET  2B  11000000        2013      2017      5
504   Bruce Rondon  DET  RP    850000        2017      2017      1
```

[55]:
```
df.columns
```

[55]:
```
Index(['NAME', 'TEAM', 'POS', 'SALARY', 'START_YEAR', 'END_YEAR', 'YEARS'],
dtype='object')
```

[56]:
```
df.shape
```

[56]:
```
(868, 7)
```

[57]:
```
df.dtypes
```

[57]:
```
NAME          object
TEAM          object
POS           object
SALARY         int64
START_YEAR     int64
END_YEAR       int64
YEARS          int64
dtype: object
```

1.0.5 Sort the data

To sort a data frame, use the `sort_values()` method. At a minimum, you need to tell it which column to sort on.

[58]:
```
df.sort_values('SALARY')
```

[58]:

	NAME	TEAM	POS	SALARY	START_YEAR	END_YEAR	YEARS
867	Yandy Diaz	CLE	3B	535000	2017	2017	1
839	Jacob May	CWS	CF	535000	2017	2017	1
838	Glenn Sparkman	TOR	RP	535000	2017	2017	1
837	Dylan Covey	CWS	RP	535000	2017	2017	1
836	Drew Robinson	TEX	OF	535000	2017	2017	1
..	…	…	..	…	…	…	…
4	Justin Verlander	DET	SP	28000000	2013	2019	7
3	Miguel Cabrera	DET	1B	28000000	2014	2023	10
2	David Price	BOS	SP	30000000	2016	2022	7
1	Zack Greinke	ARI	SP	31876966	2016	2021	6
0	Clayton Kershaw	LAD	SP	33000000	2014	2020	7

[868 rows x 7 columns]

To sort descending, you need to pass in another argument to the `sort_values()` method: `ascending=False`. Note that the boolean value is *not* a string, so it's not contained in quotes, and only the initial letter is capitalized. (If you are supplying multiple arguments to a function or method, separate them with commas.)

[59]:
```
df.sort_values('SALARY', ascending=False)
```

[59]:

	NAME	TEAM	POS	SALARY	START_YEAR	END_YEAR	YEARS
0	Clayton Kershaw	LAD	SP	33000000	2014	2020	7
1	Zack Greinke	ARI	SP	31876966	2016	2021	6
2	David Price	BOS	SP	30000000	2016	2022	7
3	Miguel Cabrera	DET	1B	28000000	2014	2023	10
4	Justin Verlander	DET	SP	28000000	2013	2019	7
..	…	…	..	…	…	…	…

```
836     Drew Robinson  TEX  OF    535000        2017       2017       1
837       Dylan Covey  CWS  RP    535000        2017       2017       1
838    Glenn Sparkman  TOR  RP    535000        2017       2017       1
839         Jacob May  CWS  CF    535000        2017       2017       1
867        Yandy Diaz  CLE  3B    535000        2017       2017       1

[868 rows x 7 columns]
```

You can use a process called "method chaining" to perform multiple operations in one line. If, for instance, we wanted to sort the data frame by salary descending and inspect the first 5 records returned:

[60]:
```
df.sort_values('SALARY', ascending=False).head()
```

[60]:
```
                NAME TEAM POS    SALARY  START_YEAR  END_YEAR  YEARS
0   Clayton Kershaw  LAD  SP  33000000        2014      2020      7
1      Zack Greinke  ARI  SP  31876966        2016      2021      6
2       David Price  BOS  SP  30000000        2016      2022      7
3    Miguel Cabrera  DET  1B  28000000        2014      2023     10
4  Justin Verlander  DET  SP  28000000        2013      2019      7
```

You can sort by multiple columns by passing in a *list* of column names rather than the name of a single column. A list is a collection of items enclosed within square brackets `[]`.

To sort first by `SALARY`, then by `TEAM`:

[61]:
```
df.sort_values(['SALARY', 'TEAM']).head()
```

[61]:
```
                  NAME TEAM POS  SALARY  START_YEAR  END_YEAR  YEARS
826     Armando Rivero  ATL  RP  535000        2017      2017      1
851      Micah Johnson  ATL  2B  535000        2017      2017      1
824  Anthony Santander  BAL  OF  535000        2017      2017      1
830         Ben Taylor  BOS  RP  535000        2017      2017      1
863       Steve Selsky  BOS  RF  535000        2017      2017      1
```

You can specify the sort order (descending vs. ascending) for each sort column by passing another list to the `ascending` keyword with `True` and `False` items corresponding to the position of the columns in the first list.

For example, to sort by `SALARY` descending, then by `TEAM` ascending:

[62]:
```
df.sort_values(['SALARY', 'TEAM'], ascending=[False, True]).head()
```

[62]:
```
                NAME TEAM POS    SALARY  START_YEAR  END_YEAR  YEARS
0   Clayton Kershaw  LAD  SP  33000000        2014      2020      7
1      Zack Greinke  ARI  SP  31876966        2016      2021      6
2       David Price  BOS  SP  30000000        2016      2022      7
3    Miguel Cabrera  DET  1B  28000000        2014      2023     10
4  Justin Verlander  DET  SP  28000000        2013      2019      7
```

The `False` goes with `SALARY` and the `True` with `TEAM` because they're in the same position in their respective lists.

One other note: Despite all of this sorting we've been doing, the original `df` data frame is unchanged:

[63]:
```
df.head()
```

[63]:

	NAME	TEAM	POS	SALARY	START_YEAR	END_YEAR	YEARS
0	Clayton Kershaw	LAD	SP	33000000	2014	2020	7
1	Zack Greinke	ARI	SP	31876966	2016	2021	6
2	David Price	BOS	SP	30000000	2016	2022	7
3	Miguel Cabrera	DET	1B	28000000	2014	2023	10
4	Justin Verlander	DET	SP	28000000	2013	2019	7

That's because we haven't "saved" the results of those sorts by assigning them to a new variable. Typically, if you want to preserve a sort (or any other kind of manipulation), you'd would assign the results to a new variable:

[64]:
```
sorted_by_team = df.sort_values('TEAM')
```

[65]:
```
sorted_by_team.head()
```

[65]:

	NAME	TEAM	POS	SALARY	START_YEAR	END_YEAR	YEARS
303	Patrick Corbin	ARI	RP	3950000	2017	2017	1
494	Chris Herrmann	ARI	C	937500	2017	2017	1
371	Fernando Rodney	ARI	RP	2500000	2017	2017	1
541	Jake Lamb	ARI	3B	573300	2017	2017	1
546	Robbie Ray	ARI	SP	570400	2017	2017	1

1.0.6 Your turn

In the cells below, practice sorting the `df` data frame: - By `NAME` - By `POS` descending - By `SALARY` descending, then by `POS` ascending, and save the results to a new variable called `sorted_by_salary_then_pos`

[ ]:
```

```

[ ]:
```

```

[ ]:
```

```

[ ]:
```

```

[ ]:
```

```

1.0.7 Filter the data

Let's go over two different kinds of filtering:

- Column filtering: Grabbing one or more columns of data to look at, like passing column names to a `SELECT` statement in SQL.
- Row filtering: Looking at a subset of your data that matches some criteria, like the crieria following a `WHERE` statement in SQL. (For instance, "Show me all records in my data frame where the value in the `TEAM` column is"ARI".)

Column filtering To access the values in a single column of data, you can use "dot notation" as long as the column name doesn't have spaces or other special characters:

[66]:
```
df.TEAM
```

[66]:
```
0      LAD
1      ARI
2      BOS
3      DET
4      DET
      …
863    BOS
864    CIN
865    LAA
866    CIN
867    CLE
Name: TEAM, Length: 868, dtype: object
```

Otherwise, use "bracket notation" with the name of the column as a string.

This is equivalent to the previous command:

[67]:
```
df['TEAM']
```

[67]:
```
0      LAD
1      ARI
2      BOS
3      DET
4      DET
      …
863    BOS
864    CIN
865    LAA
866    CIN
867    CLE
Name: TEAM, Length: 868, dtype: object
```

When you access a single column in your data frame, you're getting back something called a `Series` object (as opposed to a `DataFrame` object).

One of the methods you can call on a Series is `unique()`, which shows you each unique value in the column. Let's do that with the `TEAM` column:

[68]:
```
df.TEAM.unique()
```

[68]:
```
array(['LAD', 'ARI', 'BOS', 'DET', 'CHC', 'LAA', 'SEA', 'NYY', 'TEX',
       'SF', 'MIN', 'NYM', 'WSH', 'CIN', 'ATL', 'BAL', 'CWS', 'COL',
       'TOR', 'STL', 'MIL', 'PHI', 'HOU', 'KC', 'MIA', 'CLE', 'PIT', 'TB',
       'OAK', 'SD'], dtype=object)
```

What we just did is the equivalent of dragging the "TEAM" column name into the "rows" area of a spreadsheet pivot table, or, in SQL,

```
SELECT DISTINCT TEAM
FROM mlb
```

You can also count up a total for each value using the `value_counts()` method:

[69]:
```
df.TEAM.value_counts()
```

[69]:
```
TEX    34
TB     32
COL    32
SEA    31
BOS    31
NYM    31
SD     31
LAD    31
CIN    31
STL    30
OAK    30
LAA    30
ATL    30
TOR    29
MIN    29
MIA    28
SF     28
KC     28
BAL    28
CWS    28
ARI    28
CLE    28
NYY    27
HOU    27
WSH    26
CHC    26
PIT    26
DET    26
PHI    26
```

```
MIL    26
Name: TEAM, dtype: int64
```

For numeric columns, you can call methods on that Series to compute basic summary stats: - `min()` to get the lowest value - `max()` to get the greatest value - `median()` to get the median - `mean()` to get the average - `mode()` to get the most common value

Check it out for the `SALARY` column:

[70]:
```
df.SALARY.min()
```

[70]:
```
535000
```

[71]:
```
df.SALARY.max()
```

[71]:
```
33000000
```

[72]:
```
df.SALARY.median()
```

[72]:
```
1562500.0
```

[73]:
```
df.SALARY.mean()
```

[73]:
```
4468069.176267281
```

[74]:
```
df.SALARY.mode()
```

[74]:
```
0    535000
dtype: int64
```

To select multiple columns in your data frame, use bracket notation but pass in a *list* of column names instead of just one. To make things clearer, you could break this out into two steps:

[75]:
```
columns_we_care_about = ['TEAM', 'SALARY']
df[columns_we_care_about]
```

[75]:
```
     TEAM    SALARY
0     LAD  33000000
1     ARI  31876966
2     BOS  30000000
3     DET  28000000
4     DET  28000000
..    …        …
863   BOS    535000
864   CIN    535000
865   LAA    535000
866   CIN    535000
867   CLE    535000
```

```
[868 rows x 2 columns]
```

Row filtering To make things maximally confusing, you *also* use bracket notation for row filtering. Except in this case, instead of dropping the name of a column (or a list of column names) into the brackets, you hand it a *condition* of some sort.

Let's filter our data to see players who make more than $1 million (in other words, return rows of data where the value in the `SALARY` column is greater than 1000000):

(The equivalent SQL statement would be:

```
SELECT *
FROM mlb
WHERE SALARY > 1000000
```

)

[76]:
```
df[df.SALARY > 1000000]
```

[76]:

	NAME	TEAM	POS	SALARY	START_YEAR	END_YEAR	YEARS
0	Clayton Kershaw	LAD	SP	33000000	2014	2020	7
1	Zack Greinke	ARI	SP	31876966	2016	2021	6
2	David Price	BOS	SP	30000000	2016	2022	7
3	Miguel Cabrera	DET	1B	28000000	2014	2023	10
4	Justin Verlander	DET	SP	28000000	2013	2019	7
..	…	…	..	…	…	…	…
480	Tommy Layne	NYY	RP	1075000	2017	2017	1
481	Dan Otero	CLE	RP	1055000	2017	2017	1
482	Cory Gearrin	SF	RP	1050000	2017	2017	1
483	Kris Bryant	CHC	3B	1050000	2017	2017	1
484	Jurickson Profar	TEX	2B	1005000	2017	2017	1

[485 rows x 7 columns]

For many filters, you'll use Python's comparison operators: - `>` greater than - `>=` greater than or equal to - `<` less than - `<=` less than or equal to - `==` equal to - `!=` not equal to

Multiple filter conditions What if you want to use multiple filtering conditions? There is a way, but it usually makes more sense – and is much easier for your colleagues and your future self to think about and debug – to *save* the results of each filtering operation by assigning the results to a new variable, then filter *that* again instead of the original data frame.

For example, if you wanted to look at Colorado Rockies players who make more than $1 million, you might do something like:

[77]:
```
rockies = df[df.TEAM == 'COL']
rockies_over_1m = rockies[rockies.SALARY > 1000000]
```

[78]:
```
rockies_over_1m
```

[78]:

	NAME	TEAM	POS	SALARY	START_YEAR	END_YEAR	YEARS
31	Carlos Gonzalez	COL	RF	20428571	2011	2017	7
111	Nolan Arenado	COL	3B	11750000	2017	2018	2
155	Ian Desmond	COL	1B	8000000	2017	2021	5
158	Gerardo Parra	COL	LF	8000000	2016	2018	3
177	Charlie Blackmon	COL	CF	7300000	2017	2017	1
215	Greg Holland	COL	RP	6000000	2017	2017	1
221	Jake McGee	COL	RP	5900000	2017	2017	1
261	DJ LeMahieu	COL	2B	4800000	2016	2017	2
273	Tyler Chatwood	COL	SP	4400000	2017	2017	1
292	Mike Dunn	COL	RP	4000000	2017	2019	3
308	Chad Qualls	COL	RP	3750000	2016	2017	2
330	Jordan Lyles	COL	RP	3175000	2017	2017	1
388	Adam Ottavino	COL	RP	2100000	2016	2018	3
440	Mark Reynolds	COL	1B	1500000	2017	2017	1
460	Alexi Amarista	COL	2B	1250000	2017	2017	1

1.0.8 Your turn

In the cells below, practice filtering: - Column filtering: Select the `NAME` column - Column filtering: Select the `NAME` and `TEAM` columns - Row filtering: Filter the rows to return only players who make the league minimum (535000) - Row filtering: Filter the rows to return only catchers (`C`) who make at least 750000 - BONUS: Filter the rows to return only players for the Chicago Cubs (`CHC`), then use method chaining to order the results by `SALARY` descending

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

1.0.9 Group and aggregate the data

Data frames have a `groupby` method for grouping and aggregating data, similar to what you might do in a pivot table or a `GROUP BY` statement in SQL. (They also have a `pivot_table` method, which can be homework for you to research.)

Let's say we wanted to see the top 10 teams by payroll. In other words, we want to: - Group the data by the `TEAM` column: `groupby()` - Add up the records in each group: `sum()` - Sort the results by `SALARY` descending: `sort_values()` - Take only the top 10 results: `head(10)`

Calling the `groupby()` method without telling it what to do with the grouped records isn't super helpful:

[79]:
```
df.groupby('TEAM')
```

[79]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x11288feb0>
```

At this point, it's basically telling us that it has successfully grouped the records – now what? Using method chaining, describe what you would like to *do* with the numeric columns once you've grouped the data. Let's start with `sum()`:

[80]:
```
df.groupby('TEAM').sum()
```

[80]:

TEAM	SALARY	START_YEAR	END_YEAR	YEARS
ARI	90730499	56469	56485	44
ATL	137339527	60491	60525	64
BAL	161684185	56460	56485	53
BOS	174287098	62510	62541	62
CHC	170088502	52429	52456	53
CIN	82375785	62516	62539	54
CLE	115991166	56455	56490	63
COL	101513571	64534	64553	51
CWS	109591167	56463	56487	52
DET	180250600	52420	52457	63
HOU	127443900	54449	54469	47
KC	132091916	56459	56496	65
LAA	143965833	60493	60522	59
LAD	187989811	62501	62550	80
MIA	115348302	56470	56501	59
MIL	61020089	52430	52445	41
MIN	100787500	58478	58500	51
NYM	176284679	62511	62534	54
NYY	170389199	54439	54472	60
OAK	70601667	60503	60513	40
PHI	87418378	52436	52444	34
PIT	94637833	52428	52456	54
SD	34574400	62527	62533	37
SEA	155207720	62510	62540	61
SF	176531278	56451	56499	76
STL	146630000	60487	60532	75
TB	69982520	64537	64557	52
TEX	178431396	68555	68593	72
TOR	162353367	58478	58499	50
WSH	162742157	52421	52457	62

Neat! Except it's summing *every* numeric column, not just `SALARY`. To deal with this, use column filtering to select the two columns we're interested in – `TEAM` for grouping and `SALARY` for summing

– and *then* tack on the `groupby` statement, etc.

(Remember: To select columns from a data frame, use bracket notation and hand it a *list* of column names.)

[81]:
```
df[['TEAM', 'SALARY']].groupby('TEAM').sum()
```

[81]:

TEAM	SALARY
ARI	90730499
ATL	137339527
BAL	161684185
BOS	174287098
CHC	170088502
CIN	82375785
CLE	115991166
COL	101513571
CWS	109591167
DET	180250600
HOU	127443900
KC	132091916
LAA	143965833
LAD	187989811
MIA	115348302
MIL	61020089
MIN	100787500
NYM	176284679
NYY	170389199
OAK	70601667
PHI	87418378
PIT	94637833
SD	34574400
SEA	155207720
SF	176531278
STL	146630000
TB	69982520
TEX	178431396
TOR	162353367
WSH	162742157

Bang bang. Now, using method chaining, let's sort by `SALARY` descending and look at just the top 10:

[82]:
```
df[['TEAM', 'SALARY']].groupby('TEAM').sum().sort_values('SALARY',␣
↪ascending=False).head(10)
```

[82]:

TEAM	SALARY

```
LAD   187989811
DET   180250600
TEX   178431396
SF    176531278
NYM   176284679
BOS   174287098
NYY   170389199
CHC   170088502
WSH   162742157
TOR   162353367
```

You can use aggregation methods other than `sum()` – `mean()` and `median()`, for instance – or you can use the `agg()` method to specify one or more aggregation methods to apply.

[83]:
```
df[['TEAM', 'SALARY']].groupby('TEAM').median()
```

[83]:
```
        SALARY
TEAM
ARI   1300000
ATL   1250000
BAL   3462500
BOS   1950000
CHC   2750000
CIN    567000
CLE   2950000
COL    545000
CWS    875000
DET   1650000
HOU   3725000
KC    4000000
LAA   2250000
LAD   2600000
MIA   2762500
MIL    545950
MIN    600000
NYM   2200000
NYY   2290000
OAK    560000
PHI    562500
PIT   2962500
SD     545800
SEA   1325000
SF    2000000
STL   2762500
TB    1325000
TEX   1850000
TOR   2887500
```

```
WSH   4000000
```

[84]:
```
df[['TEAM', 'SALARY']].groupby('TEAM').mean()
```

[84]:

TEAM	SALARY
ARI	3,240,374.96
ATL	4,577,984.23
BAL	5,774,435.18
BOS	5,622,164.45
CHC	6,541,865.46
CIN	2,657,283.39
CLE	4,142,541.64
COL	3,172,299.09
CWS	3,913,970.25
DET	6,932,715.38
HOU	4,720,144.44
KC	4,717,568.43
LAA	4,798,861.10
LAD	6,064,187.45
MIA	4,119,582.21
MIL	2,346,926.50
MIN	3,475,431.03
NYM	5,686,602.55
NYY	6,310,711.07
OAK	2,353,388.90
PHI	3,362,245.31
PIT	3,639,916.65
SD	1,115,303.23
SEA	5,006,700.65
SF	6,304,688.50
STL	4,887,666.67
TB	2,186,953.75
TEX	5,247,982.24
TOR	5,598,391.97
WSH	6,259,313.73

[85]:
```
df[['TEAM', 'SALARY']].groupby('TEAM').agg(['sum', 'mean', 'median'])
```

[85]:

	SALARY		
	sum	mean	median
TEAM			
ARI	90730499	3,240,374.96	1300000
ATL	137339527	4,577,984.23	1250000
BAL	161684185	5,774,435.18	3462500
BOS	174287098	5,622,164.45	1950000
CHC	170088502	6,541,865.46	2750000

```
CIN    82375785         2,657,283.39   567000
CLE   115991166         4,142,541.64  2950000
COL   101513571         3,172,299.09   545000
CWS   109591167         3,913,970.25   875000
DET   180250600         6,932,715.38  1650000
HOU   127443900         4,720,144.44  3725000
KC    132091916         4,717,568.43  4000000
LAA   143965833         4,798,861.10  2250000
LAD   187989811         6,064,187.45  2600000
MIA   115348302         4,119,582.21  2762500
MIL    61020089         2,346,926.50   545950
MIN   100787500         3,475,431.03   600000
NYM   176284679         5,686,602.55  2200000
NYY   170389199         6,310,711.07  2290000
OAK    70601667         2,353,388.90   560000
PHI    87418378         3,362,245.31   562500
PIT    94637833         3,639,916.65  2962500
SD     34574400         1,115,303.23   545800
SEA   155207720         5,006,700.65  1325000
SF    176531278         6,304,688.50  2000000
STL   146630000         4,887,666.67  2762500
TB     69982520         2,186,953.75  1325000
TEX   178431396         5,247,982.24  1850000
TOR   162353367         5,598,391.97  2887500
WSH   162742157         6,259,313.73  4000000
```

1.0.10 Your turn

In the cells below, practice grouping data: - What's the median salary for each position? Group the data by `POS` and aggregate by `median()`, then sort by `SALARY` descending - What's the average salary on each team? Group the data by `TEAM` and aggregate by `sum()`, then sort by `SALARY` descending - What else?

[ ]:

[ ]:

[ ]:

[ ]:

1.0.11 Export to CSV

To export a dataframe to a delimited text file, use the `to_csv()` method. If you don't want to include the index numbers, specify `index=False`.

[86]:
```
df.to_csv('mlb-players-for-export.csv', index=False)
```