

# Pandas cheat sheet

January 30, 2021

## 1 Pandas cheat sheet

This notebook has some common data manipulations you might do while working in the popular Python data analysis library [pandas](#). It assumes you're already set up to analyze data in pandas using Python 3.

(If you're *not* set up, [here's IRE's guide](#) to setting up Python. [Hit me up](#) if you get stuck.)

### 1.0.1 Topics

- [Importing pandas](#)
- [Creating a dataframe from a CSV](#)
- [Checking out the data](#)
- [Selecting columns of data](#)
- [Getting unique values in a column](#)
- [Running basic summary stats](#)
- [Sorting your data](#)
- [Filtering rows of data](#)
- [Filtering text columns with string methods](#)
- [Filtering against multiple values](#)
- [Exclusion filtering](#)
- [Adding a calculated column](#)
- [Filtering for nulls](#)
- [Grouping and aggregating data](#)
- [Pivot tables](#)
- [Applying a function across rows](#)
- [Joining data](#)

### 1.0.2 Importing pandas

Before we can use pandas, we need to import it. The most common way to do this is:

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

### 1.0.3 Creating a dataframe from a CSV

To begin with, let's import a CSV of Major League Baseball player salaries on opening day. The file, which is in the same directory as this notebook, is called `mlb.csv`.

Pandas has a `read_csv()` method that we can use to get this data into a [dataframe](#) (it has methods to read other file types, too). At minimum, you need to tell this method where the file lives:

```
[82]: mlb = pd.read_csv('mlb.csv')
```

### 1.0.4 Checking out the data

When you first load up your data, you'll want to get a sense of what's in there. A pandas dataframe has several useful things to help you get a quick read of your data:

- `.head()`: Shows you the first 5 records in the data frame (optionally, if you want to see a different number of records, you can pass in a number)
- `.tail()`: Same as `head()`, but it pull records from the end of the dataframe
- `.sample(n)` will give you a sample of  $n$  rows of the data – just pass in a number
- `.info()` will give you a count of non-null values in each column – useful for seeing if any columns have null values
- `.describe()` will compute summary stats for numeric columns
- `.columns` will list the column names
- `.dtypes` will list the data types of each column
- `.shape` will give you a pair of numbers: (*number of rows*, *number of columns*)

```
[83]: mlb.head()
```

```
[83]:
```

	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

```
[84]: mlb.tail()
```

```
[84]:
```

	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

```
[85]: mlb.sample(5)
```

```
[85]:
```

	NAME	TEAM	POS	SALARY	START_YEAR	END_YEAR	YEARS
784	David Dahl	COL	CF	537000	2017	2017	1
734	Jett Bandy	MIL	C	539800	2017	2017	1
63	Wei-Yin Chen	MIA	SP	15500000	2016	2020	5
665	Kendall Graveman	OAK	SP	545000	2017	2017	1
395	Aaron Hill	SF	2B	2000000	2017	2017	1

```
[86]: mlb.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
```

```
[87]: mlb.describe()
```

```
[87]:
```

	SALARY	START_YEAR	END_YEAR	YEARS
count	8.680000e+02	868.000000	868.000000	868.000000
mean	4.468069e+06	2016.486175	2017.430876	1.944700
std	5.948459e+06	1.205923	1.163087	1.916764
min	5.350000e+05	2008.000000	2015.000000	1.000000
25%	5.455000e+05	2017.000000	2017.000000	1.000000
50%	1.562500e+06	2017.000000	2017.000000	1.000000
75%	6.000000e+06	2017.000000	2017.000000	2.000000
max	3.300000e+07	2017.000000	2027.000000	13.000000

```
[88]: mlb.columns
```

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

```
[89]: mlb.dtypes
```

```
[89]: NAME          object
      TEAM          object
      POS           object
      SALARY        int64
```

```
START_YEAR    int64
END_YEAR      int64
YEARS         int64
dtype: object
```

```
[90]: mlb.shape
```

```
[90]: (868, 7)
```

To get the number of records in a dataframe, you can access the first item in the **shape** pair, or you can just use the Python function **len()**:

```
[91]: len(mlb)
```

```
[91]: 868
```

### 1.0.5 Selecting columns of data

If you need to select just one column of data, you can use “dot notation” (**mlb.SALARY**) as long as your column name doesn’t have spaces and it isn’t the name of a dataframe method (e.g., **product**). Otherwise, you can use “bracket notation” (**mlb['SALARY']**).

Selecting one column will return a **Series**.

If you want to select multiple columns of data, use bracket notation and pass in a *list* of columns that you want to select. In Python, a list is a collection of items enclosed in square brackets, separated by commas: **['SALARY', 'NAME']**.

Selecting multiple columns will return a **DataFrame**.

```
[92]: # select one column of data
teams = mlb.TEAM

# bracket notation would do the same thing -- note the quotes around the column
↳ name
# teams = mlb['TEAM']

teams.head()
```

```
[92]: 0    LAD
      1    ARI
      2    BOS
      3    DET
      4    DET
      Name: TEAM, dtype: object
```

```
[93]: type(teams)
```

```
[93]: pandas.core.series.Series
```

```
[94]: # select multiple columns of data
salaries_and_names = mlb[['SALARY', 'NAME']]
```

```
[95]: salaries_and_names.head()
```

```
[95]:
```

	SALARY	NAME
0	33000000	Clayton Kershaw
1	31876966	Zack Greinke
2	30000000	David Price
3	28000000	Miguel Cabrera
4	28000000	Justin Verlander

```
[96]: type(salaries_and_names)
```

```
[96]: pandas.core.frame.DataFrame
```

### 1.0.6 Getting unique values in a column

As you evaluate your data, you'll often want to get a list of unique values in a column (for cleaning, filtering, grouping, etc.).

To do this, you can use the Series method `unique()`. If you wanted to get a list of baseball positions, you could do:

```
[97]: mlb.POS.unique()
```

```
[97]: array(['SP', '1B', 'RF', '2B', 'DH', 'CF', 'C', 'LF', '3B', 'SS', 'OF',
        'RP', 'P'], dtype=object)
```

If useful, you could also sort the results alphabetically with the Python `sorted()` function:

```
[98]: sorted(mlb.POS.unique())
```

```
[98]: ['1B', '2B', '3B', 'C', 'CF', 'DH', 'LF', 'OF', 'P', 'RF', 'RP', 'SP', 'SS']
```

Sometimes you just need the *number* of unique values in a column. To do this, you can use the pandas method `nunique()`:

```
[99]: mlb.POS.nunique()
```

```
[99]: 13
```

(You can also run `nunique()` on an entire dataframe:)

```
[100]: mlb.nunique()
```

```
[100]: NAME          867
      TEAM           30
      POS            13
      SALARY         419
      START_YEAR      8
      END_YEAR        10
      YEARS           11
      dtype: int64
```

If you want to count up the number of times a value appears in a column of data – the equivalent of doing a pivot table in Excel and aggregating by count – you can use the Series method `value_counts()`.

To get a list of MLB teams and the number of times each one appears in our salary data – in other words, the roster count for each team – we could do:

```
[101]: mlb.TEAM.value_counts()
```

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

Name: TEAM, dtype: int64

### 1.0.7 Running basic summary stats

Some of this already surfaced with `describe()`, but in some cases you'll want to compute these stats manually: - `sum()` - `mean()` - `median()` - `max()` - `min()`

You can run these on a Series (e.g., a column of data), or on an entire DataFrame.

```
[102]: mlb.SALARY.sum()
```

```
[102]: 3878284045
```

```
[103]: mlb.SALARY.mean()
```

```
[103]: 4468069.176267281
```

```
[104]: mlb.SALARY.median()
```

```
[104]: 1562500.0
```

```
[105]: mlb.SALARY.max()
```

```
[105]: 33000000
```

```
[106]: mlb.SALARY.min()
```

```
[106]: 535000
```

```
[107]: # entire dataframe  
mlb.mean()
```

```
[107]: SALARY      4.468069e+06  
START_YEAR    2.016486e+03  
END_YEAR      2.017431e+03  
YEARS         1.944700e+00  
dtype: float64
```

### 1.0.8 Sorting your data

You can use the `sort_values()` method to sort a dataframe by one or more columns. The default is to sort the values ascending; if you want your results sorted descending, specify `ascending=False`.

Let's sort our dataframe by SALARY descending:

```
[108]: mlb.sort_values('SALARY', ascending=False).head()
```

```
[108]:
```

		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

To sort by multiple columns, pass a list of columns to the `sort_values()` method – the sorting will happen in the order you specify in the list. You’ll also need to pass a list to the `ascending` keyword argument, otherwise both will sort ascending.

Let’s sort our dataframe first by `TEAM` ascending, then by `SALARY` descending:

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

```
[109]:
```

		NAME	TEAM	POS	SALARY	START_YEAR	END_YEAR	YEARS
1		Zack Greinke	ARI	SP	31876966	2016	2021	6
137		Yasmany Tomas	ARI	OF	9500000	2015	2020	6
149		Paul Goldschmidt	ARI	1B	8833333	2014	2018	5
190		A.J. Pollock	ARI	CF	6750000	2016	2017	2
262		Shelby Miller	ARI	SP	4700000	2017	2017	1

### 1.0.9 Filtering rows of data

To filter your data by some criteria, you’d pass your filtering condition(s) to a dataframe using bracket notation.

You can use Python’s [comparison operators](#) in your filters, which include: - > greater than - < less than - >= greater than or equal to - <= less than or equal to - == equal to - != not equal to

Example: You want to filter your data to keep records where the `TEAM` value is ‘ARI’:

```
[110]: diamondbacks = mlb[mlb.TEAM == 'ARI']
```

```
[111]: diamondbacks.head()
```

```
[111]:
```

		NAME	TEAM	POS	SALARY	START_YEAR	END_YEAR	YEARS
1		Zack Greinke	ARI	SP	31876966	2016	2021	6
137		Yasmany Tomas	ARI	OF	9500000	2015	2020	6
149		Paul Goldschmidt	ARI	1B	8833333	2014	2018	5
190		A.J. Pollock	ARI	CF	6750000	2016	2017	2
262		Shelby Miller	ARI	SP	4700000	2017	2017	1

We could filter to get all records where the `TEAM` value is *not* ‘ARI’:

```
[112]: non_diamondbacks = mlb[mlb.TEAM != 'ARI']
```

```
[113]: non_diamondbacks.head()
```



```
[113]:
```

	NAME	TEAM	POS	SALARY	START_YEAR	END_YEAR	YEARS
0	Clayton Kershaw	LAD	SP	33000000	2014	2020	7
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

We could filter our data to just grab the players that make at least \$1 million:

```
[114]: million_a_year = mlb[mlb.SALARY >= 1000000]
```

```
[115]: million_a_year.head()
```

```
[115]:
```

	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

### 1.0.10 Filtering against multiple values

You can use the `isin()` method to test a value against multiple matches – just hand it a *list* of values to check against.

Example: Let's say we wanted to filter to get just players in Texas (in other words, just the Texas Rangers and the Houston Astros):

```
[116]: tx = mlb[mlb.TEAM.isin(['TEX', 'HOU'])]
```

```
[117]: tx.head()
```

```
[117]:
```

	NAME	TEAM	POS	SALARY	START_YEAR	END_YEAR	YEARS
11	Prince Fielder	TEX	DH	24000000	2017	2017	1
15	Cole Hamels	TEX	SP	22500000	2013	2018	6
35	Shin-Soo Choo	TEX	RF	20000000	2014	2020	7
45	Adrian Beltre	TEX	3B	18000000	2017	2018	2
52	Brian McCann	HOU	C	17000000	2014	2018	5

### 1.0.11 Exclusion filtering

Sometimes it's easier to specify what records you *don't* want returned. To flip the meaning of a filter condition, prepend a tilde `~`.

For instance, if we wanted to get all players who are *not* from Texas, we'd use the same filter condition we just used to get the TX players but add a tilde at the beginning:

```
[118]: not_tx = mlb[~mlb.TEAM.isin(['TEX', 'HOU'])]
```

```
[119]: not_tx.head()
```

```
[119]:
```

	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

### 1.0.12 Filtering text columns with string methods

You can access the text values in a column with `.str`, and you can use any of Python's native string functions to manipulate them.

For our purposes, though, the pandas `str.contains()` method is useful for filtering data by matching text patterns.

If we wanted to get every player with 'John' in their name, we could do something like this:

```
[120]: johns = mlb[mlb.NAME.str.contains('John', case=False)]
```

```
[121]: johns.head()
```

```
[121]:
```

	NAME	TEAM	POS	SALARY	START_YEAR	END_YEAR	YEARS
12	Johnny Cueto	SF	SP	23500000	2016	2021	6
60	John Lackey	CHC	SP	16000000	2016	2017	2
237	John Axford	OAK	RP	5500000	2016	2017	2
255	Jim Johnson	ATL	RP	5000000	2017	2018	2
295	John Jaso	PIT	1B	4000000	2016	2017	2

Note the `case=False` keyword argument – we're telling pandas to match case-insensitive. And if the pattern you're trying to match is more complex, the method is set up to support [regular expressions](#) by default.

### 1.0.13 Multiple filters

Sometimes you have multiple filters to apply to your data. Lots of the time, it makes sense to break the filters out into separate statements.

For instance, if you wanted to get all Texas players who make at least \$1 million, I might do this:

```
[122]: tx = mlb[mlb.TEAM.isin(['TEX', 'HOU'])]

# note that I'm filtering the dataframe I just created, not the original `mlb`
↳ dataframe
```

```
tx_million_a_year = tx[tx.SALARY >= 1000000]
```

```
[123]: tx_million_a_year.head()
```

```
[123]:
```

	NAME	TEAM	POS	SALARY	START_YEAR	END_YEAR	YEARS
11	Prince Fielder	TEX	DH	24000000	2017	2017	1
15	Cole Hamels	TEX	SP	22500000	2013	2018	6
35	Shin-Soo Choo	TEX	RF	20000000	2014	2020	7
45	Adrian Beltre	TEX	3B	18000000	2017	2018	2
52	Brian McCann	HOU	C	17000000	2014	2018	5

But sometimes you want to chain your filters together into one statement. Use `|` for “or” and `&` for “and” rather than Python’s built-in `or` and `and` statements, and use grouping parentheses around each statement.

The same filter in one statement:

```
[124]: tx_million_a_year = mlb[(mlb.TEAM.isin(['TEX', 'HOU'])) & (mlb.SALARY > ↵  
↵1000000)]
```

```
[125]: tx_million_a_year.head()
```

```
[125]:
```

	NAME	TEAM	POS	SALARY	START_YEAR	END_YEAR	YEARS
11	Prince Fielder	TEX	DH	24000000	2017	2017	1
15	Cole Hamels	TEX	SP	22500000	2013	2018	6
35	Shin-Soo Choo	TEX	RF	20000000	2014	2020	7
45	Adrian Beltre	TEX	3B	18000000	2017	2018	2
52	Brian McCann	HOU	C	17000000	2014	2018	5

Do what works for you and makes sense in context, but I find the first version a little easier to read.

### 1.0.14 Adding a calculated column

To add a new column to a dataframe, use bracket notation to supply the name of the new column (in quotes, or apostrophes, as long as they match), then set it equal to a value – maybe a calculation derived from other data in your dataframe.

For example, let’s create a new column, `contract_total`, that multiplies the annual salary by the number of contract years:

```
[126]: mlb['contract_total'] = mlb['SALARY'] * mlb['YEARS']
```

```
[127]: mlb.head()
```

```
[127]:
```

	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

	contract_total
0	231000000
1	191261796
2	210000000
3	280000000
4	196000000

### 1.0.15 Filtering for nulls

You can use the `isnull()` method to get records that are null, or `notnull()` to get records that aren't. The most common use I've seen for these methods is during filtering to see how many records you're missing (and, therefore, how that affects your analysis).

The MLB data is complete, so to demonstrate this, let's load up a new data set: A cut of the [National Inventory of Dams](#) database, courtesy of the NICAR data library. (We'll need to specify the `encoding` on this CSV because it's not UTF-8.)

```
[128]: dams = pd.read_csv('dams.csv',
                        encoding='latin-1')
```

```
[129]: dams.head()
```

```
[129]:
```

	NIDID	Dam_Name	Insp_Date	Submit_Date	\
0	VA16104	CLIFFORD D. CRAIG MEMORIAL DAM	2007-09-06	2013-03-12	
1	VA07915	GREENE MOUNTAIN LAKE DAM	2008-07-14	2013-03-12	
2	VA06906	LEHMANS DAM	NaN	2013-03-12	
3	VA13905	LURAY	2010-12-22	2013-02-28	
4	VA06106	MATHEWS DAM	NaN	2013-03-12	

  

	River	City_02	County	State	Cong_Dist	\
0	TRIB. TO ROANOKE RIVER	SALEM	ROANOKE CO	VA	VA09	
1	BLUE RUN	ADVANCE MILLS	GREENE	VA	VA05	
2	GOUGH RUN	MARLBORO	FREDERICK	VA	VA10	
3	SOUTH FORK SHENANDOAH RIVER	RILEYVILLE	PAGE	VA	VA06	
4	TR-GAP RUN	RECTORTOWN	FAUQUIER	VA	VA05	

  

	Cong_Rep	...	Fed_Fund	Fed_Design	Fed_Con	Fed_Reg	Fed_Insp	\
0	H. MORGAN GRIFFITH (R)	...	NaN	NaN	NaN	NaN	NaN	
1	ROBERT HURT (R)	...	NaN	NaN	NaN	NaN	NaN	
2	FRANK R. WOLF (R)	...	NaN	NaN	NaN	NaN	NaN	
3	BOB GOODLATTE (R)	...	NaN	NaN	NaN	FERC	FERC	
4	ROBERT HURT (R)	...	NaN	NaN	NaN	NaN	NaN	

	Srce_Agency	Oth_StrucID	Num_Struc	Longitude	Latitude
0	VA	NaN	0	-80.1750	37.2250
1	VA	NaN	0	-78.4366	38.2700
2	VA	NaN	0	-78.3083	39.1516
3	FERC	NaN	1	-78.4999	38.6774
4	VA	NaN	0	-77.9600	38.9800

[5 rows x 42 columns]

Maybe we're interested in looking at the year the dam was completed (the `Year_Comp`) column. Running `.info()` on the dataframe shows that we're missing some values:

```
[130]: dams.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2482 entries, 0 to 2481
Data columns (total 42 columns):
#   Column          Non-Null Count  Dtype
---  -
0   NIDID            2482 non-null   object
1   Dam_Name         2480 non-null   object
2   Insp_Date        1093 non-null   object
3   Submit_Date      2482 non-null   object
4   River            2264 non-null   object
5   City_02          1407 non-null   object
6   County           2477 non-null   object
7   State            2482 non-null   object
8   Cong_Dist        2445 non-null   object
9   Cong_Rep         2445 non-null   object
10  Party            2445 non-null   object
11  Owner_Type       2482 non-null   object
12  Owner_Name       2199 non-null   object
13  Year_Comp         1663 non-null   float64
14  Year_Mod          438 non-null    object
15  Private_Dam      2482 non-null   object
16  NPDP_Hazard      1487 non-null   object
17  Permit_Auth      2482 non-null   object
18  Insp_Auth        2482 non-null   object
19  Enfrc_Auth       2482 non-null   object
20  Juris_Dam        2482 non-null   object
21  NID_Height       2468 non-null   float64
22  NID_Storage      2453 non-null   float64
23  Dam_Length       1813 non-null   float64
24  Max_Discharge    831 non-null    float64
25  Drain_Area       1188 non-null   float64
26  Dam_Designer     831 non-null    object
27  EAP              2482 non-null   object
28  Insp_Freq        2482 non-null   int64
```

```

29 St_Reg_Dam      2482 non-null  object
30 St_Reg_Agency  2374 non-null  object
31 Volume         530 non-null  float64
32 Fed_Fund       219 non-null  object
33 Fed_Design     578 non-null  object
34 Fed_Con        221 non-null  object
35 Fed_Reg        132 non-null  object
36 Fed_Insp       146 non-null  object
37 Srce_Agency    2482 non-null  object
38 Oth_StrucID    17 non-null   object
39 Num_Struc      2482 non-null  int64
40 Longitude      2482 non-null  float64
41 Latitude       2482 non-null  float64
dtypes: float64(9), int64(2), object(31)
memory usage: 814.5+ KB

```

We can filter for `isnull()` to take a closer look:

```
[131]: no_year_comp = dams[dams.Year_Comp.isnull()]
```

```
[132]: no_year_comp.head()
```

```
[132]:
```

	NIDID		Dam_Name	Insp_Date	Submit_Date	\
43	DE00095		WAPLES POND DAM	NaN	2013-02-04	
114	VA17710		LEE LAKE DAM	2003-09-08	2013-03-12	
152	VA19104	HIDDEN VALLEY LAKE DAM		2004-12-31	2013-03-12	
212	MD00018		EMMITSBURG DAM	2012-08-30	2013-02-04	
263	DE00070		BLAIRS POND DAM	2011-07-26	2013-02-04	

  

	River		City_02		County	State	Cong_Dist	\
43	PRIMEHOOK CREEK	BROADKILL BEACH	E		SUSSEX	DE	DE00	
114	WILDERNESS RUN		MINE RUN		SPOTSYLVANIA	VA	VA07	
152	BRUMLEY CREEK		DUNCANVILLE		WASHINGTON	VA	VA09	
212	TURKEY CREEK		EMMITSBURG		FREDERICK	MD	MD08	
263	BEAVERDAM BRANCH		HOUSTON		KENT	DE	DE00	

  

		Cong_Rep	...	Fed_Fund	Fed_Design	Fed_Con	Fed_Reg	\
43	JOHN C. CARNEY JR.	(D)	...	NaN	NaN	NaN	NaN	
114	ERIC CANTOR	(R)	...	NaN	NaN	NaN	NaN	
152	H. MORGAN GRIFFITH	(R)	...	NaN	NaN	NaN	NaN	
212	CHRIS VAN HOLLEN	(D)	...	NaN	NaN	NaN	NaN	
263	JOHN C. CARNEY JR.	(D)	...	NaN	NaN	NaN	NaN	

  

	Fed_Insp	Srce_Agency	Oth_StrucID	Num_Struc	Longitude	Latitude
43	NaN	DE	NaN	0	-75.3087	38.8240
114	NaN	VA	NaN	0	-77.7400	38.3033
152	NaN	VA	NaN	0	-82.0733	36.8500
212	NaN	MD	NaN	0	-77.3885	39.6959

```
263      NaN      DE      NaN      0 -75.4848  38.9039
```

```
[5 rows x 42 columns]
```

How many are we missing? That will help us determine whether the analysis would be valid:

```
[133]: # calculate the percentage of records with no Year_Comp value
# (part / whole) * 100

(len(no_year_comp) / len(dams)) * 100
```

```
[133]: 32.99758259468171
```

So this piece of our analysis would exclude one-third of our records – something you’d need to explain to your audience, if indeed your reporting showed that the results of your analysis would still be meaningful.

To get records where the `Year_Comp` is not null, we’d use `notnull()`:

```
[134]: has_year_comp = dams[dams.Year_Comp.notnull()]
```

```
[135]: has_year_comp.head()
```

```
[135]:
```

	NIDID	Dam_Name	Insp_Date	Submit_Date	\
0	VA16104	CLIFFORD D. CRAIG MEMORIAL DAM	2007-09-06	2013-03-12	
1	VA07915	GREENE MOUNTAIN LAKE DAM	2008-07-14	2013-03-12	
2	VA06906	LEHMANS DAM	NaN	2013-03-12	
3	VA13905	LURAY	2010-12-22	2013-02-28	
4	VA06106	MATHEWS DAM	NaN	2013-03-12	

  

	River	City_02	County	State	Cong_Dist	\
0	TRIB. TO ROANOKE RIVER	SALEM	ROANOKE CO	VA	VA09	
1	BLUE RUN	ADVANCE MILLS	GREENE	VA	VA05	
2	GOUGH RUN	MARLBORO	FREDERICK	VA	VA10	
3	SOUTH FORK SHENANDOAH RIVER	RILEYVILLE	PAGE	VA	VA06	
4	TR-GAP RUN	RECTORTOWN	FAUQUIER	VA	VA05	

  

	Cong_Rep	...	Fed_Fund	Fed_Design	Fed_Con	Fed_Reg	Fed_Insp	\
0	H. MORGAN GRIFFITH (R)	...	NaN	NaN	NaN	NaN	NaN	
1	ROBERT HURT (R)	...	NaN	NaN	NaN	NaN	NaN	
2	FRANK R. WOLF (R)	...	NaN	NaN	NaN	NaN	NaN	
3	BOB GOODLATTE (R)	...	NaN	NaN	NaN	FERC	FERC	
4	ROBERT HURT (R)	...	NaN	NaN	NaN	NaN	NaN	

  

	Srce_Agncy	Oth_StrucID	Num_Struc	Longitude	Latitude
0	VA	NaN	0	-80.1750	37.2250
1	VA	NaN	0	-78.4366	38.2700
2	VA	NaN	0	-78.3083	39.1516

3	FERC	NaN	1	-78.4999	38.6774
4	VA	NaN	0	-77.9600	38.9800

[5 rows x 42 columns]

What years remain? Let's use `value_counts()` to find out:

```
[136]: has_year_comp.Year_Comp.value_counts()
```

```
[136]: 1960.0    86
      1965.0    56
      1974.0    54
      1955.0    52
      1967.0    51
      ..
      1832.0     1
      1914.0     1
      1682.0     1
      1922.0     1
      1881.0     1
      Name: Year_Comp, Length: 142, dtype: int64
```

(To sort by year, not count, we could tack on a `sort_index()`:

```
[137]: has_year_comp.Year_Comp.value_counts().sort_index()
```

```
[137]: 1682.0     1
      1694.0     1
      1780.0     2
      1800.0    11
      1801.0     1
      ..
      2008.0     7
      2009.0     6
      2010.0     2
      2011.0     1
      2012.0     1
      Name: Year_Comp, Length: 142, dtype: int64
```

### 1.0.16 Grouping and aggregating data

You can use the `groupby()` method to group and aggregate data in pandas, similar to what you'd get by running a pivot table in Excel or a `GROUP BY` query in SQL. We'll also provide the aggregate function to use.

Let's group our baseball salary data by team to see which teams have the biggest payrolls – in other words, we want to use `sum()` as our aggregate function:



```
[138]: grouped_mlb = mlb.groupby('TEAM').sum()
```

```
[139]: grouped_mlb.head()
```

```
[139]:
```

	SALARY	START_YEAR	END_YEAR	YEARS	contract_total
TEAM					
ARI	90730499	56469	56485	44	341698661
ATL	137339527	60491	60525	64	593579662
BAL	161684185	56460	56485	53	510234644
BOS	174287098	62510	62541	62	749308534
CHC	170088502	52429	52456	53	648189802

If you don't specify what columns you want, it will run `sum()` on every numeric column. Typically I select just the grouping column and the column I'm running the aggregation on:

```
[140]: grouped_mlb = mlb[['TEAM', 'SALARY']].groupby('TEAM').sum()
```

```
[141]: grouped_mlb.head()
```

```
[141]:
```

	SALARY
TEAM	
ARI	90730499
ATL	137339527
BAL	161684185
BOS	174287098
CHC	170088502

... and we can sort descending, with `head()` to get the top payrolls:

```
[142]: grouped_mlb.sort_values('SALARY', ascending=False).head(10)
```

```
[142]:
```

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

You can use different aggregate functions, too. Let's say we wanted to get the top median salaries by team:

```
[143]: mlb[['TEAM', 'SALARY']].groupby('TEAM').median().sort_values('SALARY',
↪ascending=False).head(10)
```

```
[143]:          SALARY
TEAM
WSH    4000000
KC     4000000
HOU    3725000
BAL    3462500
PIT    2962500
CLE    2950000
TOR    2887500
STL    2762500
MIA    2762500
CHC    2750000
```

You can group by multiple columns by passing a list. Here, we'll select our columns of interest and group by `TEAM`, then by `POS`, using `sum()` as our aggregate function:

```
[144]: mlb[['TEAM', 'POS', 'SALARY']].groupby(['TEAM', 'POS']).sum()
```

```
[144]:          SALARY
TEAM POS
ARI  1B    10183333
      3B     1127200
      C     4437500
      CF     7289500
      LF      542500
...
WSH  LF    22971429
      RF    13625000
      RP    15698700
      SP    54886428
      SS      537800
```

```
[306 rows x 1 columns]
```

### 1.0.17 Pivot tables

Sometimes you need a full-blown pivot table, and [pandas has a function to make one](#).

For this example, we'll look at some foreign trade data – specifically, eel product imports from 2010 to mid-2017:

```
[145]: eels = pd.read_csv('eels.csv')
```

```
[146]: eels.head()
```

```
[146]:   year  month  country    product  kilos  dollars
      0  2010      1    CHINA  EELS FROZEN  49087   393583
      1  2010      1    JAPAN  EELS FRESH    263     7651
      2  2010      1   TAIWAN  EELS FROZEN   9979   116359
      3  2010      1  VIETNAM  EELS FRESH   1938    10851
      4  2010      1  VIETNAM  EELS FROZEN  21851    69955
```

Let's run a pivot table where the grouping column is `country`, the values are the sum of `kilos`, and the columns are the year:

```
[147]: pivoted_sums = pd.pivot_table(eels,
                                     index='country',
                                     columns='year',
                                     values='kilos',
                                     aggfunc=sum)
```

```
[148]: pivoted_sums.head()
```

```
[148]: year          2010      2011      2012      2013      2014      2015  \
country
BANGLADESH      NaN      NaN      13.0      NaN      NaN      600.0
BURMA           NaN      NaN      NaN      NaN      NaN      NaN
CANADA         13552.0  24968.0  110796.0  44455.0  31546.0  28619.0
CHILE           NaN      NaN      NaN      NaN      6185.0      NaN
CHINA          372397.0  249232.0  1437392.0  1090135.0  1753140.0  4713882.0

year          2016      2017
country
BANGLADESH      NaN      NaN
BURMA           699.0      NaN
CANADA         68568.0  23571.0
CHILE           NaN      NaN
CHINA          4578546.0  1771272.0
```

Let's sort by the 2017 value. While we're at it, let's fill in null values (`NaN`) with zeroes using the `fillna()` method.

```
[149]: pivoted_sums.sort_values(2017, ascending=False).fillna(0)
```

```
[149]: year          2010      2011      2012      2013      2014  \
country
CHINA          372397.0  249232.0  1437392.0  1090135.0  1753140.0
TAIWAN          73842.0      0.0   53774.0   39752.0   83478.0
SOUTH KOREA     42929.0  41385.0   28146.0   27353.0   37708.0
JAPAN           1326.0   2509.0   32255.0  105758.0   40177.0
THAILAND        2866.0   5018.0    9488.0    4488.0   15110.0
VIETNAM         63718.0  155488.0  118063.0  100828.0   38112.0
CANADA         13552.0  24968.0   110796.0   44455.0   31546.0
```

PORTUGAL	2081.0	3672.0	2579.0	2041.0	7215.0
PANAMA	0.0	0.0	0.0	11849.0	0.0
BANGLADESH	0.0	0.0	13.0	0.0	0.0
BURMA	0.0	0.0	0.0	0.0	0.0
CHILE	0.0	0.0	0.0	0.0	6185.0
CHINA - HONG KONG	0.0	0.0	0.0	0.0	0.0
COSTA RICA	0.0	0.0	0.0	0.0	0.0
INDIA	0.0	0.0	0.0	0.0	0.0
MEXICO	0.0	0.0	0.0	4000.0	0.0
NEW ZEALAND	0.0	2652.0	900.0	270.0	0.0
NORWAY	0.0	0.0	0.0	17391.0	0.0
PAKISTAN	0.0	0.0	0.0	22453.0	0.0
PHILIPPINES	0.0	0.0	0.0	610.0	0.0
POLAND	0.0	0.0	1296.0	0.0	864.0
SENEGAL	0.0	1350.0	0.0	0.0	0.0
SPAIN	0.0	0.0	977.0	275.0	1019.0
UKRAINE	0.0	0.0	0.0	0.0	0.0

year	2015	2016	2017
country			
CHINA	4713882.0	4578546.0	1771272.0
TAIWAN	48272.0	99535.0	44087.0
SOUTH KOREA	8386.0	14729.0	42904.0
JAPAN	69699.0	71748.0	37892.0
THAILAND	41771.0	26931.0	31884.0
VIETNAM	36859.0	96179.0	28490.0
CANADA	28619.0	68568.0	23571.0
PORTUGAL	8013.0	9105.0	6747.0
PANAMA	0.0	0.0	974.0
BANGLADESH	600.0	0.0	0.0
BURMA	0.0	699.0	0.0
CHILE	0.0	0.0	0.0
CHINA - HONG KONG	0.0	735.0	0.0
COSTA RICA	0.0	563.0	0.0
INDIA	0.0	2200.0	0.0
MEXICO	16860.0	0.0	0.0
NEW ZEALAND	0.0	0.0	0.0
NORWAY	0.0	0.0	0.0
PAKISTAN	0.0	0.0	0.0
PHILIPPINES	0.0	0.0	0.0
POLAND	0.0	0.0	0.0
SENEGAL	0.0	0.0	0.0
SPAIN	719.0	1008.0	0.0
UKRAINE	0.0	11414.0	0.0

### 1.0.18 Applying a function across rows

Often, you'll want to calculate a value for every column but it won't be that simple, and you'll write a separate function that accepts one row of data, does some calculations and returns a value. We'll use the `apply()` method to accomplish this.

For this example, we're going to load up a CSV of gators killed by hunters in Florida:

```
[150]: gators = pd.read_csv('gators.csv')
```

```
[151]: gators.head()
```

```
[151]:
```

	Year	Area Number	Area Name	Carcass Size	Harvest Date	Location
0	2000	101	LAKE PIERCE	11 ft. 5 in.	09-22-2000	
1	2000	101	LAKE PIERCE	9 ft. 0 in.	10-02-2000	
2	2000	101	LAKE PIERCE	8 ft. 10 in.	10-06-2000	
3	2000	101	LAKE PIERCE	8 ft. 0 in.	09-25-2000	
4	2000	101	LAKE PIERCE	8 ft. 0 in.	10-07-2000	

We want to find the longest gator in our data, of course, but there's a problem: right now, the carcass size value is being stored as text: `{ } ft. { } in.`. The pattern is predicatable, though, and we can use some Python to turn those values into constant numbers – inches – that we can then sort on. Here's our function:

```
[152]: def get_inches(row):  
    '''Accepts a row from our dataframe, calculates carcass length in inches,  
    and returns that value'''  
  
    # get the value in the 'Carcass Size' column  
    carcass_size = row['Carcass Size']  
  
    # split the text on 'ft.'  
    # the result is a list  
    size_split = carcass_size.split('ft.')  
  
    # strip whitespace from the first item ([0]) in the resulting list -- the  
    # feet --  
    # and coerce it to an integer with the Python `int()` function  
    feet = int(size_split[0].strip())  
  
    # in the second item ([1]) in the resulting list -- the inches -- replace  
    # 'in.' with nothing,  
    # strip whitespace and coerce to an integer  
    inches = int(size_split[1].replace('in.', '').strip())  
  
    # add the feet times 12 plus the inches and return that value  
    return inches + (feet * 12)
```

Now we're going to create a new column, `length_in` and use the `apply()` method to apply our

function to every row. The `axis=1` keyword argument means that we're applying our function row-wise, not column-wise.

```
[153]: gators['length_in'] = gators.apply(get_inches, axis=1)
```

```
[154]: gators.sort_values('length_in', ascending=False).head()
```

```
[154]:
```

	Year	Area Number	Area Name	Carcass Size	\
44996	2010	502	ST. JOHNS RIVER (LAKE POINSETT)	14 ft. 3 in.	
78315	2014	828	HIGHLANDS COUNTY	14 ft. 3 in.	
31961	2008	510	LAKE JESUP	14 ft. 1 in.	
70005	2013	733	LAKE TALQUIN	14 ft. 1 in.	
63077	2012	828	HIGHLANDS COUNTY	14 ft. 0 in.	

	Harvest Date	Location	length_in
44996	10-31-2010		171
78315	10-28-2014	LITTLE RED WATER LAKE	171
31961	08-26-2008		169
70005	09-02-2013		169
63077	10-31-2012	boat ramp north of boat ramp road	168

### 1.0.19 Joining data

You can use `merge()` to join data in pandas.

In this simple example, we're going to take a CSV of country population data in which each country is represented by an [ISO 3166-1 numeric country code](#) and join it to a CSV that's basically a lookup table with the ISO codes and the names of the countries to which they refer.

Some of the country codes have leading zeroes, so we're going to use the `dtype` keyword when we import each CSV to specify that the 'code' column in each dataset should be treated as a string (text), not a number.

```
[155]: pop_csv = pd.read_csv('country-population.csv', dtype={'code': str})
```

```
[156]: pop_csv.head()
```

```
[156]:
```

	code	pop2000	pop2001	pop2002	pop2003	pop2004	pop2005	pop2006	\
0	108	6401.0	6556.0	6742.0	6953.0	7182.0	7423.0	7675.0	
1	174	542.0	556.0	569.0	583.0	597.0	612.0	626.0	
2	262	718.0	733.0	746.0	759.0	771.0	783.0	796.0	
3	232	3393.0	3497.0	3615.0	3738.0	3859.0	3969.0	4067.0	
4	231	66537.0	68492.0	70497.0	72545.0	74624.0	76727.0	78851.0	

  

	pop2007	pop2008	pop2009	pop2010	pop2011	pop2012	pop2013	pop2014	\
0	7940.0	8212.0	8489.0	8767.0	9044.0	9320.0	9600.0	9892.0	
1	642.0	657.0	673.0	690.0	707.0	724.0	742.0	759.0	
2	809.0	823.0	837.0	851.0	866.0	881.0	897.0	912.0	

```

3  4153.0  4233.0  4310.0  4391.0  4475.0  4561.0  4651.0  4746.0
4  81000.0  83185.0  85416.0  87703.0  90047.0  92444.0  94888.0  97367.0

```

```

      pop2015
0  10199.0
1    777.0
2    927.0
3   4847.0
4  99873.0

```

```
[157]: code_csv = pd.read_csv('country-codes.csv', dtype={'code': str})
```

```
[158]: code_csv.head()
```

```
[158]:   code  country
0   108  Burundi
1   174  Comoros
2   262  Djibouti
3   232  Eritrea
4   231  Ethiopia

```

Now we'll use `merge()` to join them.

The `on` keyword argument tells the method what column to join on. If the names of the columns were different, you'd use `left_on` and `right_on`, with the “left” dataframe being the first one you hand to the `merge()` function.

The `how` keyword argument tells the method what type of join to use – the default is `'inner'`.

```
[159]: joined_data = pd.merge(pop_csv,
                             code_csv,
                             on='code',
                             how='left')
```

```
[160]: joined_data.head()
```

```
[160]:   code  pop2000  pop2001  pop2002  pop2003  pop2004  pop2005  pop2006  \
0   108   6401.0   6556.0   6742.0   6953.0   7182.0   7423.0   7675.0
1   174    542.0    556.0    569.0    583.0    597.0    612.0    626.0
2   262    718.0    733.0    746.0    759.0    771.0    783.0    796.0
3   232   3393.0   3497.0   3615.0   3738.0   3859.0   3969.0   4067.0
4   231  66537.0  68492.0  70497.0  72545.0  74624.0  76727.0  78851.0

      pop2007  pop2008  pop2009  pop2010  pop2011  pop2012  pop2013  pop2014  \
0   7940.0   8212.0   8489.0   8767.0   9044.0   9320.0   9600.0   9892.0
1    642.0    657.0    673.0    690.0    707.0    724.0    742.0    759.0
2    809.0    823.0    837.0    851.0    866.0    881.0    897.0    912.0
3   4153.0   4233.0   4310.0   4391.0   4475.0   4561.0   4651.0   4746.0

```

4 81000.0 83185.0 85416.0 87703.0 90047.0 92444.0 94888.0 97367.0

	pop2015	country
0	10199.0	Burundi
1	777.0	Comoros
2	927.0	Djibouti
3	4847.0	Eritrea
4	99873.0	Ethiopia