Recap

- Introduced DataFrame concepts
  - Series: A named column of data with an index
  - Indexes: The mapping from keys to rows
  - DataFrame: collection of series with common index

- Dataframe access methods
  - [ ]: bracket operator for selecting columns or rows
  - df.loc: location by index
  - df.iloc: location by integer address

Pandas (Continued)

- How to filter data based on certain criteria?
  - Boolean array selection and querying
  - Examples of indexing with .loc and .iloc
- How to randomly sample data?
  - Sampling with .sample
- Useful utility functions for Series and DataFrames
  - Example: sort_values, head, size, describe

Boolean Array Selection and Querying

Boolean Array Input

Yet another input type supported by [ ] is the boolean array

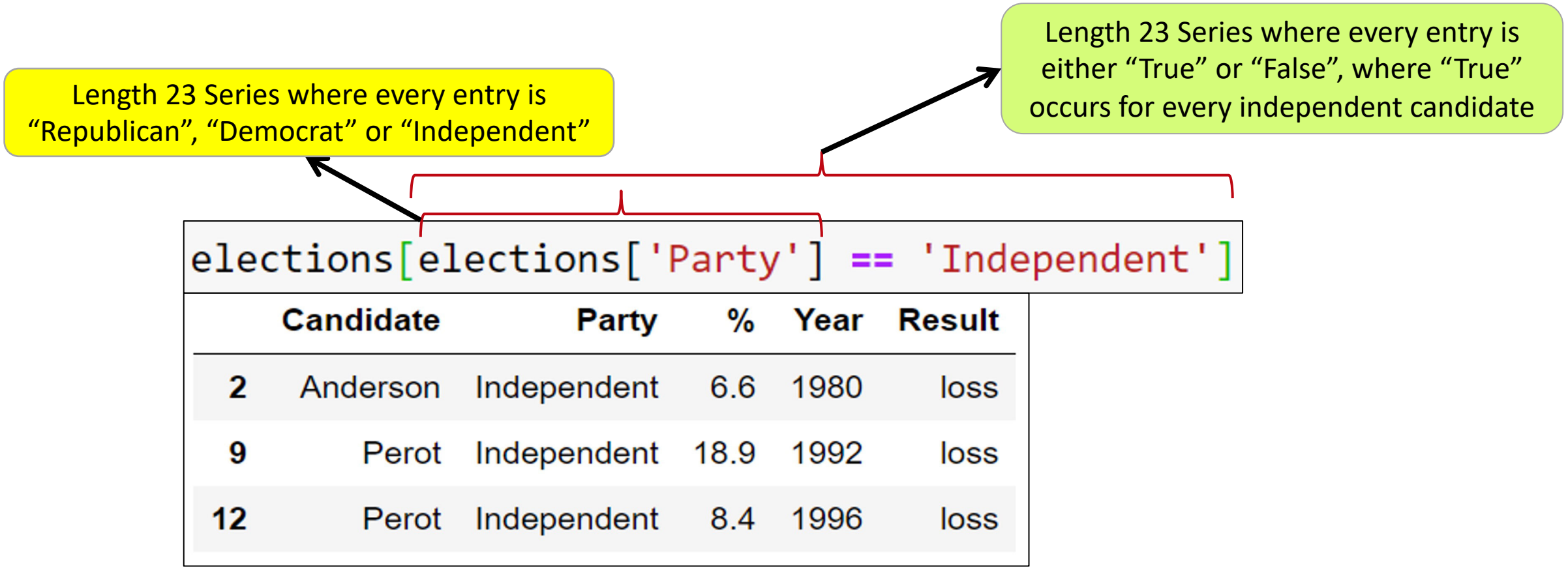
Entry number 7

```
elections[[False, False, False, False, False,
           False, False, True, False, False,
           True, False, False, False, True,
           False, False, False, False, False,
           False, False, True]]
```

	Candidate	Party	%	Year	Result
7	Clinton	Democratic	43.0	1992	win
10	Clinton	Democratic	49.2	1996	win
14	Bush	Republican	47.9	2000	win
22	Trump	Republican	46.1	2016	win

Boolean Array Input

Yet another input type supported by [ ] is the boolean array. Useful because boolean arrays can be generated by using logical operators on Series



	Candidate	Party	%	Year	Result
2	Anderson	Independent	6.6	1980	loss
9	Perot	Independent	18.9	1992	loss
12	Perot	Independent	8.4	1996	loss

Boolean Array Input

Boolean Series can be combined using the & operator, allowing filtering of results by multiple criteria

```
elections[(elections['Result'] == 'win')
          & (elections['%'] < 50)]
```

	Candidate	Party	%	Year	Result
7	Clinton	Democratic	43.0	1992	win
10	Clinton	Democratic	49.2	1996	win
14	Bush	Republican	47.9	2000	win
22	Trump	Republican	46.1	2016	win

Alternate Approaches for Filtering

- The **isin** function makes it convenient to find rows that match one of many possible values

```
df[(df["Party"] == "Democratic") | (df["Party"] == "Republican")]
```

vs.

```
df[df["Party"].isin(["Republican", "Democratic"])]
```

- The **query** command provides an alternate way to combine multiple conditions

```
elections.query("Result == 'win' and Year < 2000")
```

- See previous lecture for details

Indexing with .loc and .iloc
Sampling with .sample

loc and iloc

loc and iloc are alternate ways to index into a DataFrame

- They take a lot of getting used to! Documentation and ideas behind them are quite complex
- I'll go over common usages (see docs for weirder ones ☺)

Documentation:

[1] loc: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.loc.html

[2] iloc: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.iloc.html

[3] More general docs on indexing and selecting: Link

Locator

loc does two things:

- Access values by labels
- Access values using a boolean array (a.k.a. Boolean Array Selection)

Loc with Lists as input

The most basic use of loc is to provide a list of row and column labels, which returns a DataFrame

```
elections.loc[[0, 1, 2, 3, 4], ['Candidate','Party', 'Year']]
```

	Candidate	Party	Year
0	Reagan	Republican	1980
1	Carter	Democratic	1980
2	Anderson	Independent	1980
3	Reagan	Republican	1984
4	Mondale	Democratic	1984

Loc with Lists as input

The most basic use of loc is to provide a list of row and column labels, which returns a DataFrame

```
elections_year_index.loc[[1980, 1984], ['Candidate','Party']]
```

Year	Candidate	Party
1980	Reagan	Republican
1980	Carter	Democratic
1980	Anderson	Independent
1984	Reagan	Republican
1984	Mondale	Democratic

Example with year as the new index

Loc with Slices as input

Loc is also commonly used with slices

- Slicing works with all label types, not just numeric labels
- Slices with loc are inclusive, not exclusive

```
elections.loc[0:4, 'Candidate':'Year']
```

	Candidate	Party	Year
0	Reagan	Republican	1980
1	Carter	Democratic	1980
2	Anderson	Independent	1980
3	Reagan	Republican	1984
4	Mondale	Democratic	1984

Loc with Single Values for Column Label

If we provide only a single label as column argument, we get a Series

```
elections.loc[0:4, 'Candidate']
```

```
0       Reagan
1       Carter
2    Anderson
3       Reagan
4      Mondale
Name: Candidate, dtype: object
```

Loc with Single Values for Column Label

As before with the [ ] operator, if we provide a list of only one label as an argument, we get back a DataFrame

```
elections.loc[0:4, 'Candidate']
```

```
0       Reagan
1       Carter
2    Anderson
3       Reagan
4      Mondale
Name: Candidate, dtype: object
```

```
elections.loc[0:4, ['Candidate']]
```

	Candidate
0	Reagan
1	Carter
2	Anderson
3	Reagan
4	Mondale

Loc with Single Values for Row Label

If we provide only a single row label, we get a Series

- Such a series represents a Row not a column!
- The index of this Series is the names of the columns from the data frame
- Putting the single row label in a list yields a DataFrame version

```
elections.loc[0, 'Candidate':'Year']
```

```
Candidate       Reagan
Party       Republican
%                 50.7
Year              1980
Name: 0, dtype: object
```

```
elections.loc[[0], 'Candidate':'Year']
```

	Candidate	Party	%	Year
0	Reagan	Republican	50.7	1980

Loc Supports Boolean Arrays

Loc supports Boolean Arrays exactly as you'd expect

```
elections.loc[(elections['Result'] == 'win') & (elections['%'] < 50), 'Candidate':'%']
```

	Candidate	Party	%
7	Clinton	Democratic	43.0
10	Clinton	Democratic	49.2
14	Bush	Republican	47.9
22	Trump	Republican	46.1

iloc: Integer-Based Indexing for Selection by Position

In contrast to loc, iloc doesn't think about labels at all. Instead, it returns the items that appear in the numerical positions specified

```
elections.iloc[0:3, 0:3]
```

	Candidate	Party	%
0	Reagan	Republican	50.7
1	Carter	Democratic	41.0
2	Anderson	Independent	6.6

```
mottos.iloc[0:3, 0:3]
```

State	Motto	Translation	Language
Alabama	Audemus jura nostra defendere	We dare defend our rights!	Latin
Alaska	North to the future	—	English
Arizona	Ditat Deus	God enriches	Latin

Advantages of loc:

- Harder to make mistakes
- Easier to read code
- Not vulnerable to changes to the ordering of rows/cols in raw data files

Nonetheless, iloc can be more convenient. Use iloc judiciously

Poll

Question

Which of the following pandas statements returns a DataFrame containing data about the first 3 Candidates only, who won with more than 50% of the votes. Only the "Candidate" and "Year" columns should be part of the DataFrame.

A. `elections.loc[[0, 3, 5], ["Candidate":"Year"]]`

B. `elections.iloc[[0, 3, 5], [0, 3]]`

C. `elections.loc[elections["%"] > 50, ["Candidate", "Year"]].head(3)`

D. `elections.loc[elections["%"] > 50, ["Candidate", "Year"]].iloc[0:2, :]`

	Candidate	Party	%	Year	Result
0	Reagan	Republican	50.7	1980	win
1	Carter	Democratic	41.0	1980	loss
2	Anderson	Independent	6.6	1980	loss
3	Reagan	Republican	58.8	1984	win
4	Mondale	Democratic	37.6	1984	loss
5	Bush	Republican	53.4	1988	win
6	Dukakis	Democratic	45.6	1988	loss

	Candidate	Year
0	Reagan	1980
3	Reagan	1984
5	Bush	1988

Poll

Question

Which of the following pandas statements returns a DataFrame containing data about the first 3 Candidates only, who won with more than 50% of the votes. Only the "Candidate" and "Year" columns should be part of the DataFrame.

A. `elections.loc[[0, 3, 5], ["Candidate":"Year"]]`

B. `elections.iloc[[0, 3, 5], [0, 3]]`

C. `elections.loc[elections["%"] > 50, ["Candidate", "Year"]].head(3)`

D. `elections.loc[elections["%"] > 50, ["Candidate", "Year"]].iloc[0:2, :]`

	Candidate	Party	%	Year	Result
0	Reagan	Republican	50.7	1980	win
1	Carter	Democratic	41.0	1980	loss
2	Anderson	Independent	6.6	1980	loss
3	Reagan	Republican	58.8	1984	win
4	Mondale	Democratic	37.6	1984	loss
5	Bush	Republican	53.4	1988	win
6	Dukakis	Democratic	45.6	1988	loss

	Candidate	Year
0	Reagan	1980
3	Reagan	1984
5	Bush	1988

See notebook for why!

Random Sampling using Sample

If you want a DataFrame consisting of a random selection of rows, you can use the sample method

- By default, it is without replacement. Use `replace=true` for replacement
- Can be chained with our selection operators [ ], loc, iloc, query, etc

```
elections.sample(10)
```

	Candidate	Party	%	Year	Result
15	Kerry	Democratic	48.3	2004	loss
16	Bush	Republican	50.7	2004	win
22	Trump	Republican	46.1	2016	win
9	Perot	Independent	18.9	1992	loss
21	Clinton	Democratic	48.2	2016	loss
11	Dole	Republican	40.7	1996	loss
20	Romney	Republican	47.2	2012	loss
14	Bush	Republican	47.9	2000	win
8	Bush	Republican	37.4	1992	loss
1	Carter	Democratic	41.0	1980	loss

```
elections.query("Year < 1992").sample(4, replace=True)
```

	Candidate	Party	%	Year	Result
1	Carter	Democratic	41.0	1980	loss
4	Mondale	Democratic	37.6	1984	loss
6	Dukakis	Democratic	45.6	1988	loss
1	Carter	Democratic	41.0	1980	loss

Handy Properties and Utility Functions for Series and DataFrames

Numpy (or Numerical Python) Operations

Pandas Series and DataFrames support a large number of operations, including mathematical operations so long as the data is numerical

```
1  winners = elections.loc[elections['Result']=='win']['%']
2  winners
```

```
0      50.7
3      58.8
5      53.4
7      43.0
10     49.2
14     47.9
16     50.7
17     52.9
19     51.1
22     46.1
Name: %, dtype: float64
```

```
np.mean(winners)
```

```
50.38
```

```
max(winners)
```

```
58.8
```

`head`, `size`, `shape`, and `describe`

`head`: Displays only the top few rows
`size`: Gives the total number of data points
`shape`: Gives the size of the data in rows and columns
`describe`: Provides a summary of the data

`index` and `columns`

`index`: Returns the index (a.k.a. row labels)

`columns`: Returns the labels for the columns

The `sort_values` Method

- One incredibly useful method for DataFrames is `sort_values`, which creates a copy of a DataFrame sorted by a specific column

```
elections.sort_values('%', ascending=False)
```

	Candidate	Party	%	Year	Result
3	Reagan	Republican	58.8	1984	win
5	Bush	Republican	53.4	1988	win
17	Obama	Democratic	52.9	2008	win
19	Obama	Democratic	51.1	2012	win
0	Reagan	Republican	50.7	1980	win

The `sort_values` Method

- We can also use `sort_values` on a Series, which returns a copy with the values in order

```
mottos['Language'].sort_values().head(5)
```

```
State
Washington          Chinook Jargon
Wyoming                    English
New Jersey                 English
New Hampshire              English
Nevada                     English
Name: Language, dtype: object
```

The `value_counts` Method

- Series also has the function `value_counts`, which creates a new Series showing the counts of every value

```
elections['Party'].value_counts()
```

```
Democratic     10
Republican     10
Independent     3
Name: Party, dtype: int64
```

The `unique` Method

- Another handy method for Series is **`unique`**, which returns all unique values as an array

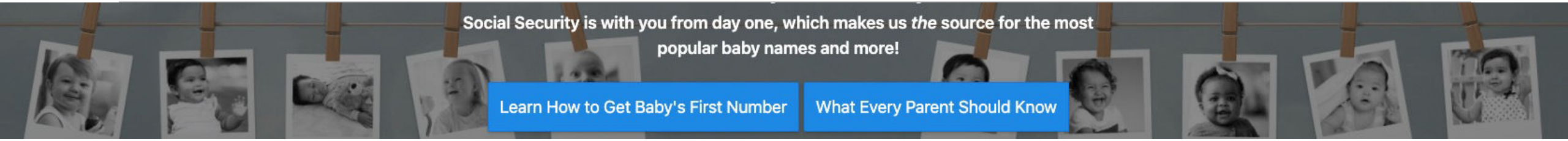
```
mottos['Language'].unique()
```

```
array(['Latin', 'English', 'Greek', 'Hawaiian', 'Italian', 'French',
       'Spanish', 'Chinook Jargon'], dtype=object)
```

The Things We Just Saw

- sort_values
- value_counts
- unique

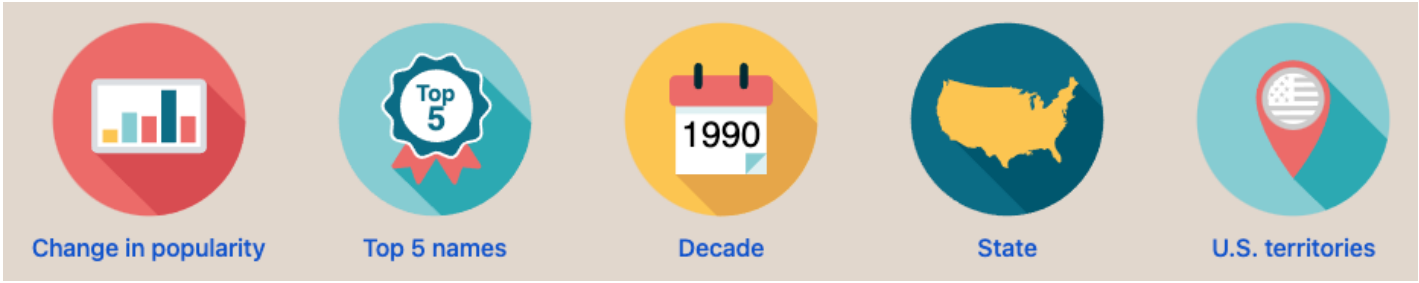
Baby Names Dataset



To provide *popular* names and maintain an acceptable performance level on our servers, we provide only the top 1000 names through our forms. However, we provide almost all names for researchers interested in naming trends.

To safeguard privacy, we exclude from these files certain names that would indicate, or would allow the ability to determine, names with fewer than 5 occurrences in any geographic area. We provide these data on both a national and state-specific basis, in two separate collections of files, each zipped into a single file. The format of the data in the three file collections is described in a "*readme*" file contained in the respective zip files.

- National data (7Mb)
- State-specific data (21Mb)
- Territory-specific data (300Kb)



Source: https://www.ssa.gov/oact/babynames/limits.html

Outline

We'll introduce additional syntax by trying to solve various practical problems on our baby names dataset

- Goal 1: Find the most popular name in California in 2012
- Goal 2: Find all names that start with J
- Goal 3: Sort names by length
- Goal 4: Find the name whose popularity has changed the most
- Goal 5: Count the number of female and male babies born in each year

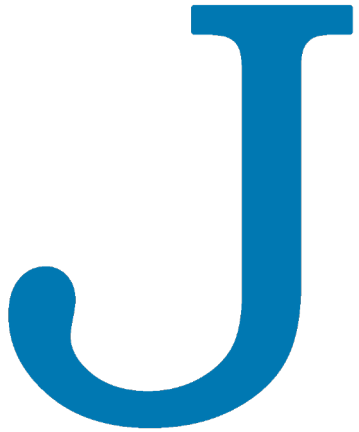
Questions on the baby names dataset

- Goal 1: Find the most popular name in California in 2012
- Goal 2: Find all names that start with J
- Goal 3: Sort names by length
- Goal 4: Find the name whose popularity has changed the most
- Goal 5: Count the number of female and male babies born in each year

Let’s go the notebook

Questions on the baby names dataset

- Goal 1: Find the most popular name in California in 2012
- Goal 2: Find all names that start with J
- Goal 3: Sort names by length
- Goal 4: Find the name whose popularity has changed the most
- Goal 5: Count the number of female and male babies born in each year



str

Str Methods

The `str` methods from the `Series` class have fairly intuitive behavior

Example: `str.startswith`

```
babynames[babynames["Name"].str.startswith('J')].sample(5)
```

	State	Sex	Year	Name	Count
32151	CA	F	1953	Jewel	11
316051	CA	M	1995	Jarrett	32
344242	CA	M	2006	Josemanuel	26
343769	CA	M	2006	Junior	96
172550	CA	F	2006	Jazmin	582

Str Methods

The `str` methods from the `Series` class have intuitive behavior

Example: `str.contains`

```
babynames[babynames["Name"].str.contains('ad')].sample(5)
```

	State	Sex	Year	Name	Count
221233	CA	F	2018	Madelyn	336
221518	CA	F	2018	Guadalupe	98
290499	CA	M	1984	Bradford	32
152534	CA	F	2000	Khadija	5
132159	CA	F	1995	Soledad	31

Str Methods

The `str` methods from the `Series` class have fairly intuitive behavior

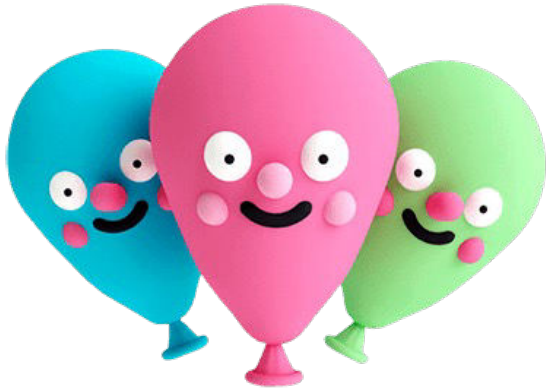
Example: `str.split`

```
babynames["Name"].str.split('a').to_frame().head(5)
```

	Name
0	[M, ry]
1	[Helen]
2	[Dorothy]
3	[M, rg, ret]
4	[Fr, nces]

Questions on the baby names dataset

- Goal 1: Find the most popular name in California in 2012
- Goal 2: Find all names that start with J
- Goal 3: Sort names by length
- Goal 4: Find the name whose popularity has changed the most
- Goal 5: Count the number of female and male babies born in each year



Sorting By Length

Goal 3: Sort baby names by length

The `sort_values` function does not provide the ability to pass a custom comparison function

Let's see two different ways of doing this that are much nicer

- Approach 1: Creating a temporary column, then sort on it
- Approach 2: Creating a sorted index and using loc

Adding, Modifying, and Removing Columns

Approach 1: Create a Temporary Column

Intuition: Create a column equal to the length. Sort by that column

	State	Sex	Year	Name	Count	name_lengths
312731	CA	M	1993	Ryanchristopher	5	15
322558	CA	M	1997	Franciscojavier	5	15
297806	CA	M	1987	Franciscojavier	5	15
307174	CA	M	1991	Franciscojavier	6	15
302145	CA	M	1989	Franciscojavier	6	15

Syntax for Column Addition

Adding a column is easy:

```
#create a new series of only the lengths
babyname_lengths = babynames["Name"].str.len()

#add that series to the dataframe as a column
babynames["name_lengths"] = babyname_lengths
```

Can also do both steps on one line of code

	State	Sex	Year	Name	Count	name_lengths
0	CA	F	1910	Mary	295	4
1	CA	F	1910	Helen	239	5
2	CA	F	1910	Dorothy	220	7
3	CA	F	1910	Margaret	163	8
4	CA	F	1910	Frances	134	7

Syntax for Dropping a Column (or Row)

After sorting, we can drop the temporary column

- The Drop method assumes you're dropping a row by default
  Use axis = 1 to drop a column instead

```
babynames = babynames.drop("name_lengths", axis = 1)
```

	State	Sex	Year	Name	Count	name_lengths
312731	CA	M	1993	Ryanchristopher	5	15
322558	CA	M	1997	Franciscojavier	5	15
297806	CA	M	1987	Franciscojavier	5	15
307174	CA	M	1991	Franciscojavier	6	15
302145	CA	M	1989	Franciscojavier	6	15

	State	Sex	Year	Name	Count
312731	CA	M	1993	Ryanchristopher	5
322558	CA	M	1997	Franciscojavier	5
297806	CA	M	1987	Franciscojavier	5
307174	CA	M	1991	Franciscojavier	6
302145	CA	M	1989	Franciscojavier	6

Approach 2: Create a Sorted Index and Pass to .loc

Another approach is to take advantage of a feature of .loc

- `df.loc[idx]` returns the DataFrame in the same order as the given index
- Only works if the index exactly matches the DataFrame

Let's see this approach in action

Approach 2: Create a Sorted Index and Pass to .loc

Step 1: Create a series of only the lengths of the names

- This Series will have the same index as the original DataFrame

```
name_lengths = babynames["Name"].str.len()
name_lengths.head(5)
```

	State	Sex	Year	Name	Count
340748	CA	M	2005	Pedro	442
294382	CA	M	1986	Royce	32
241809	CA	M	1943	Les	7
52043	CA	F	1965	Cristine	18
308476	CA	M	1992	Reyes	24

babynames

```
340748     5
294382     5
241809     3
52043      8
308476     5
Name: Name, dtype: int64
```

name_lengths

Approach 2: Create a Sorted Index and Pass to .loc

Step 2: Sort the series of only name lengths

- This Series will have an index which is reordered relative to the original dataframe

```
name_lengths_sorted_by_length = name_lengths.sort_values()
name_lengths_sorted_by_length.head(5)
```

```
340748    5
294382    5
241809    3
52043     8
308476    5
Name: Name, dtype: int64
```

name_lengths

```
111450    2
165876    2
57212     2
307201    2
329408    2
Name: Name, dtype: int64
```

name_lengths_sorted_by_length

Approach 2: Create a Sorted Index and Pass to .loc

Step 3: Pass the sorted index as an argument of .loc to the original dataframe

```
index_sorted_by_length = name_lengths_sorted_by_length.index
babynames.loc[index_sorted_by_length].head(5)
```

```
111450    2
165876    2
57212     2
307201    2
329408    2
Name: Name, dtype: int64
```

name_lengths_sorted_by_length

	State	Sex	Year	Name	Count
111450	CA	F	1989	Vy	8
165876	CA	F	2004	An	17
57212	CA	F	1968	Jo	80
307201	CA	M	1991	Jc	6
329408	CA	M	2000	Al	7

babynames.loc[index_sorted_by_length]