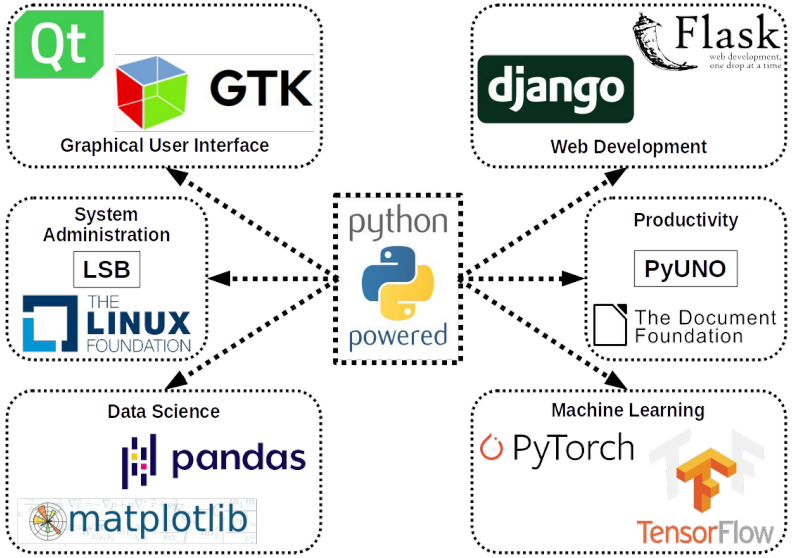
Outline

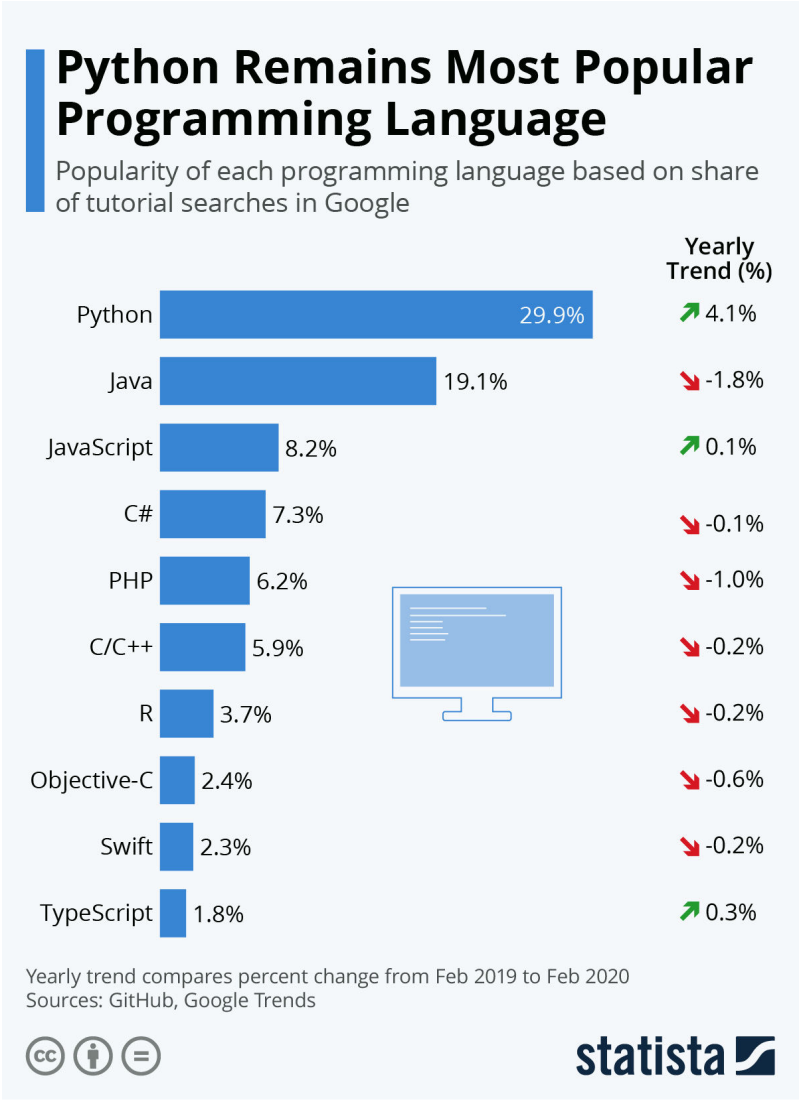
- Introduce Pandas, with emphasis on:
  - A statistical view of DataFrames
  - Key data structures: data frames, series, indices
  - How to index into these structures?
  - How to read files to create these structures?
  - Other basic operations on these structures
- Will go through a lot of the language without full explanations
  - We expect you to fill in the gaps on assignments, project, and through your own experimentation
- Solve some basic data science problems using Jupyter/pandas

Python Programming Langauge

- Designed by Guido van Rossum in 1991
- A high-level, general-purpose language
  - Places emphasis code readability
  - Dynamically-typed and garbage-collected
  - Supports multiple paradigms (e.g., procedural, object-oriented)



Python Growth



IEEE SPECTRUM

Rank	Language	Type	Score
1	Python		100.0
2	Java		95.3
3	C		94.6
4	C++		87.0
5	JavaScript		79.5
6	R		78.6
7	Arduino		73.2
8	Go		73.1

Data Frames: a high-level, statistical perspective

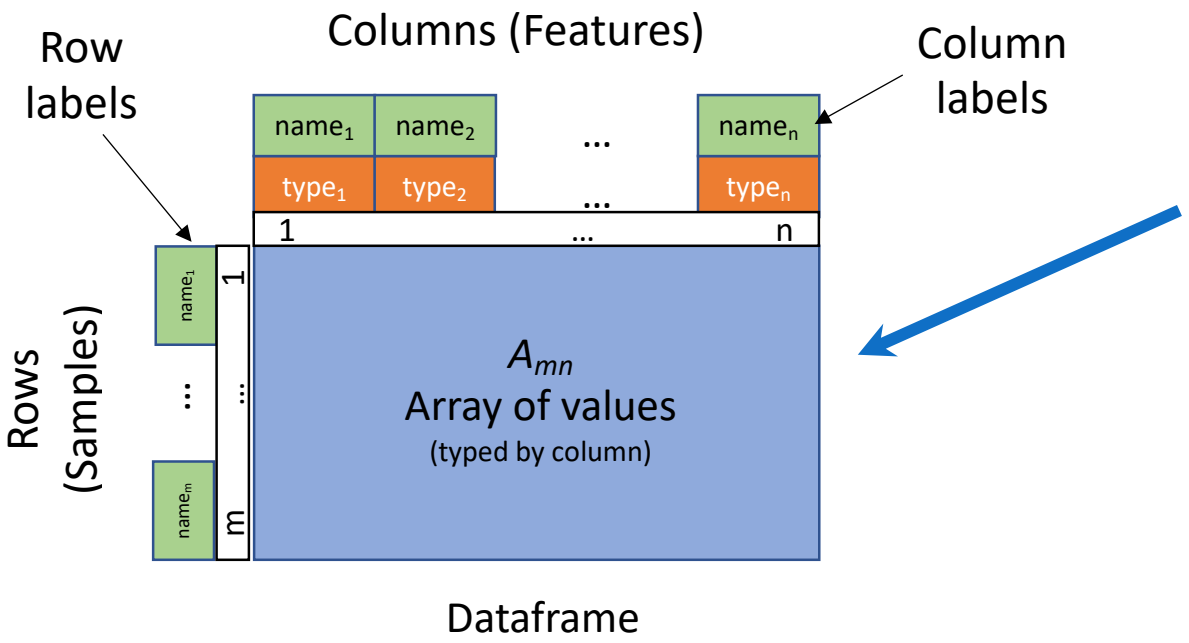
A Statistician's View of the World

We samples from a population
Each sample has certain features

Sample

Sample

	Candidate	Party	%	Year	Result
0	Obama	Democratic	52.9	2008	win
1	McCain	Republican	45.7	2008	loss



	Candidate	Party	%	Year	Result
0	Obama	Democratic	52.9	2008	win
1	McCain	Republican	45.7	2008	loss
2	Obama	Democratic	51.1	2012	win
3	Romney	Republican	47.2	2012	loss
4	Clinton	Democratic	48.2	2016	loss
5	Trump	Republican	46.1	2016	win

A Theory of Dataframes

Towards Scalable Dataframe Systems

Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo
Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, Aditya Parameswaran
UC Berkeley

{devin.petersohn, smacke, dorx, williamma, dorislee, xmo, jegonzal, hellerstein, adj, adityagp} @berkeley.edu

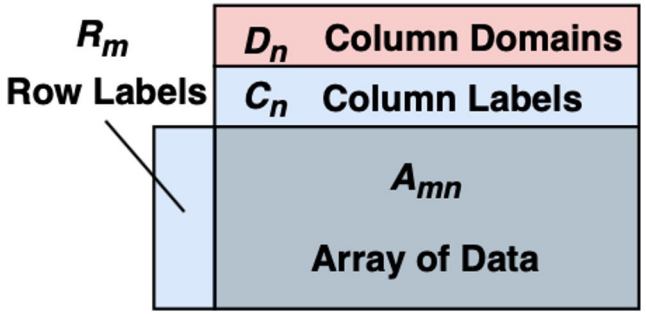
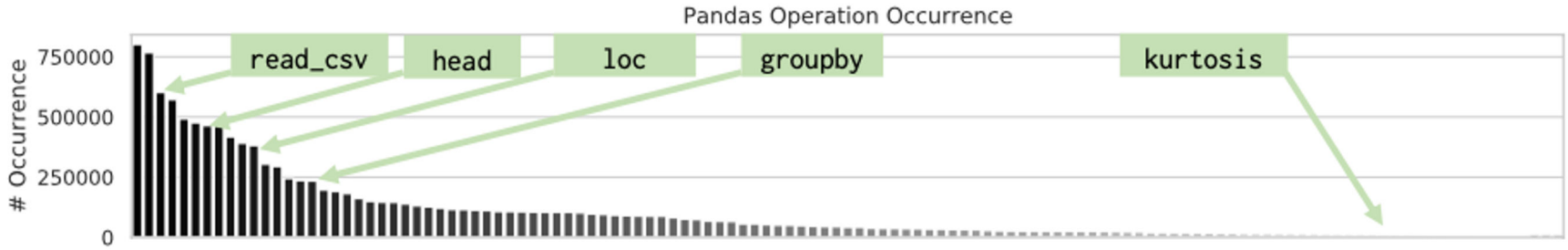


Figure 4: The Dataframe Data Model

ABSTRACT

Dataframes are a popular abstraction to represent, prepare, and analyze data. Despite the remarkable success of dataframe libraries in R and Python, dataframes face performance issues even on moderately large datasets. Moreover, there is significant ambiguity regarding dataframe semantics. In this paper we lay out a vision and roadmap for scalable dataframe systems. To demonstrate the potential in this area, we report on our experience building MODIN, a scaled-up implementation of the most widely-used and complex dataframe API today, Python's pandas. With pandas as a reference, we propose a simple data model and algebra for dataframes to ground discussion in the field. Given this foundation, we lay out an agenda of open research opportunities where the distinct features of dataframes will require extending the state of the art in many dimensions of data management. We discuss the implications of signature dataframe features including flexible schemas, ordering, row/column equivalence, and data/metadata fluidity, as well as the piecemeal, trial-and-error-based approach to interacting with dataframes.



https://arxiv.org/abs/2001.00888

Figure 7: Pandas user statistics from GitHub dataset.

Pandas Data Structures:
Data Frames, Series, and Indices

Pandas Data Structures

There are three fundamental data structures in pandas:

- Data Frame: 2D tabular data
- Series: 1D data. I usually think of it as columnar data
- Index: A sequence of row labels

Data Frame

	Candidate	Party	%	Year	Result
0	Obama	Democratic	52.9	2008	win
1	McCain	Republican	45.7	2008	loss
2	Obama	Democratic	51.1	2012	win
3	Romney	Republican	47.2	2012	loss
4	Clinton	Democratic	48.2	2016	loss
5	Trump	Republican	46.1	2016	win

Series

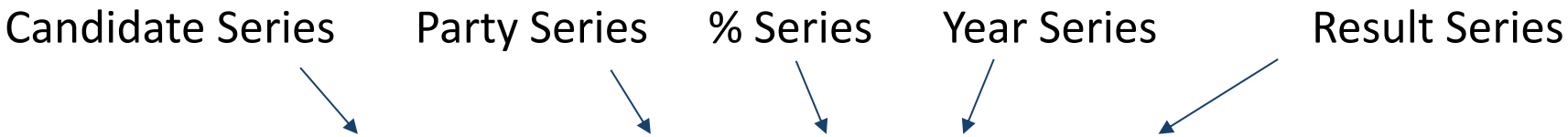
```
0       Obama
1      McCain
2       Obama
3      Romney
4     Clinton
5       Trump
Name: Candidate, dtype: object
```

Index

Relation between Data Frames, Series, and Indices

We can think of a Data Frame as a collection of Series all sharing the same Index

- Candidate, Party, %, Year, and Result Series all share an index from 0 to 5



	Candidate	Party	%	Year	Result
0	Obama	Democratic	52.9	2008	win
1	McCain	Republican	45.7	2008	loss
2	Obama	Democratic	51.1	2012	win
3	Romney	Republican	47.2	2012	loss
4	Clinton	Democratic	48.2	2016	loss
5	Trump	Republican	46.1	2016	win

Indices Are Not Necessarily Row Numbers

Indices (a.k.a. row labels) can also:

- Be non-numeric
- Have a name, e.g. “State”

State	Motto	Translation	Language	Date Adopted
Alabama	Audemus jura nostra defendere	We dare defend our rights!	Latin	1923
Alaska	North to the future	—	English	1967
Arizona	Ditat Deus	God enriches	Latin	1863
Arkansas	Regnat populus	The people rule	Latin	1907
California	Eureka (Εὕρηκα)	I have found it	Greek	1849

Indices

The row labels that constitute an index do not have to be unique

- Left: The index values are all unique and numeric, acting as a row number
- Right: The index values are named and non-unique

	Candidate	Party	%	Year	Result
0	Obama	Democratic	52.9	2008	win
1	McCain	Republican	45.7	2008	loss
2	Obama	Democratic	51.1	2012	win
3	Romney	Republican	47.2	2012	loss
4	Clinton	Democratic	48.2	2016	loss
5	Trump	Republican	46.1	2016	win

	Candidate	Party	%	Result
Year				
2008	Obama	Democratic	52.9	win
2008	McCain	Republican	45.7	loss
2012	Obama	Democratic	51.1	win
2012	Romney	Republican	47.2	loss
2016	Clinton	Democratic	48.2	loss
2016	Trump	Republican	46.1	win

Column Names Are Usually Unique!

Column names in Pandas are almost always unique!

- Example: Really shouldn't have two columns named "Candidate"

	Candidate	Party	%	Year	Result
0	Obama	Democratic	52.9	2008	win
1	McCain	Republican	45.7	2008	loss
2	Obama	Democratic	51.1	2012	win
3	Romney	Republican	47.2	2012	loss
4	Clinton	Democratic	48.2	2016	loss
5	Trump	Republican	46.1	2016	win

Structure of a Series/DataFrames

Internal indicies

	idx	temp
0	0	23
1	1	24
2	2	26
3	4	23

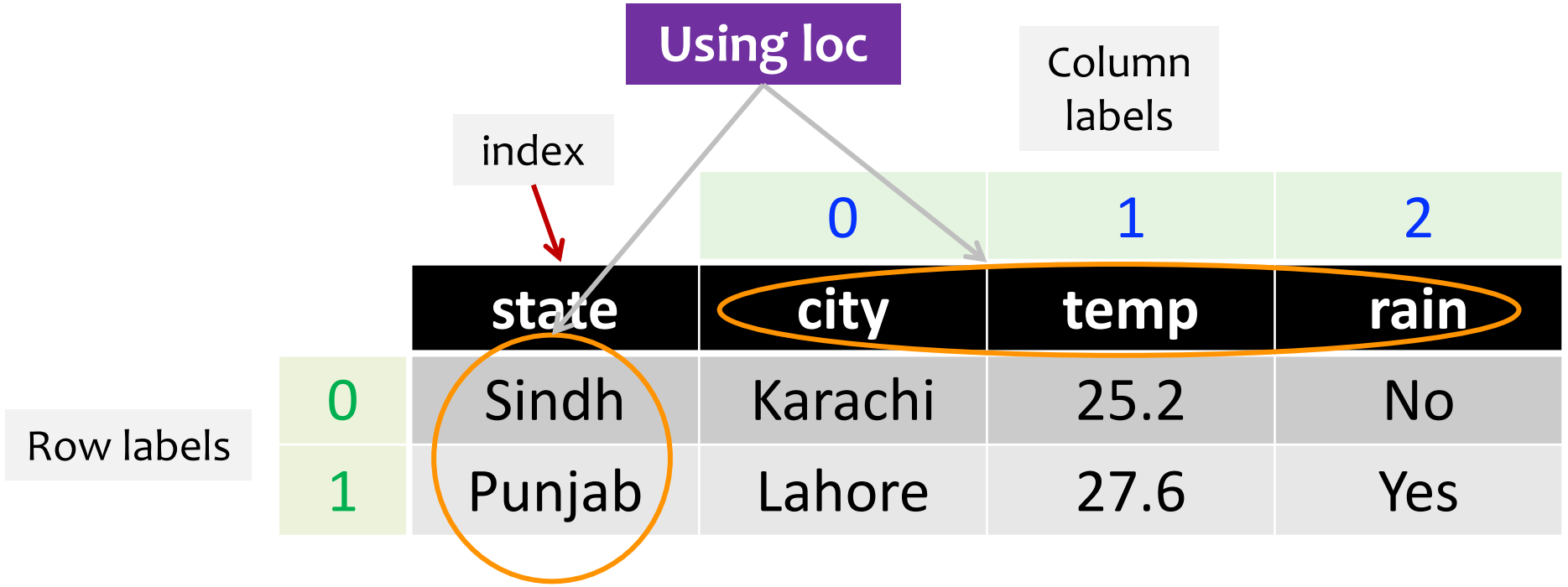
Access:
iloc: integer/internal
loc: labels (even if integer)

index

		0	1	2
	state	city	temp	rain
0	Sindh	Karachi	25.2	No
1	Punjab	Lahore	27.6	Yes

dtypes: String object, float

Structure of a Series/DataFrames



Hands On Exercise

	A	B	C	D	E
1	Candidate	Party	%	Year	Result
2	Reagan	Republican	50.7	1980	win
3	Carter	Democratic	41	1980	loss
4	Anderson	Independent	6.6	1980	loss
5	Reagan	Republican	58.8	1984	win
6	Mondale	Democratic	37.6	1984	loss
7	Bush	Republican	53.4	1988	win
8	Dukakis	Democratic	45.6	1988	loss
9	Clinton	Democratic	43	1992	win
10	Bush	Republican	37.4	1992	loss
11	Perot	Independent	18.9	1992	loss
12	Clinton	Democratic	49.2	1996	win
13	Dole	Republican	40.7	1996	loss
14	Perot	Independent	8.4	1996	loss
15	Gore	Democratic	48.4	2000	loss
16	Bush	Republican	47.9	2000	win
17	Kerry	Democratic	48.3	2004	loss
18	Bush	Republican	50.7	2004	win
19	Obama	Democratic	52.9	2008	win
20	McCain	Republican	45.7	2008	loss
21	Obama	Democratic	51.1	2012	win
22	Romney	Republican	47.2	2012	loss
23	Clinton	Democratic	48.2	2016	loss
24	Trump	Republican	46.1	2016	win

Let's experiment with reading csv files and playing around with indices

- See lec4-pandas-basics.ipynb

Indexing with the [ ] "Brack" Operator

Indexing by Column Names Using [ ] Operator

Given a dataframe, it is common to extract a Series or a collection of Series. This process is also known as "column selection" or sometimes "indexing by column"

- Column name argument to [ ] yields a Series
- List argument to [ ] yields a Data Frame

```
elections["Candidate"].head(6)
```

```
Year
1980       Reagan
1980       Carter
1980    Anderson
1984       Reagan
1984      Mondale
1988         Bush
Name: Candidate, dtype: object
```

```
elections[["Candidate", "Party"]].head(6)
```

Year	Candidate	Party
1980	Reagan	Republican
1980	Carter	Democratic
1980	Anderson	Independent
1984	Reagan	Republican
1984	Mondale	Democratic
1988	Bush	Republican

Indexing by Column Names Using [ ] Operator

Given a dataframe, it is common to extract a Series or a collection of Series. This process is also known as "column selection" or sometimes "indexing by column"

- Column name argument to [ ] yields Series
- List argument (even of one name) to [ ] yields a Data Frame

```
elections["Candidate"].head(6)
```

```
Year
1980       Reagan
1980       Carter
1980    Anderson
1984       Reagan
1984      Mondale
1988         Bush
Name: Candidate, dtype: object
```

```
elections[["Candidate"]].head(6)
```

Year	Candidate
1980	Reagan
1980	Carter
1980	Anderson
1984	Reagan
1984	Mondale
1988	Bush

Indexing by Row Slices Using [ ] Operator

We can also index by row numbers using the [ ] operator

- Numeric slice argument to [ ] yields rows
- Example: [0:3] yields rows 0 to 2

```
elections[0:3]
```

Year	Candidate	Party	%	Result
1980	Reagan	Republican	50.7	win
1980	Carter	Democratic	41.0	loss
1980	Anderson	Independent	6.6	loss

[ ] Summary

Name → [ ] → Series

Single Column Selection

```
elections["Candidate"].head(6)
```

```
Year
1980       Reagan
1980       Carter
1980    Anderson
1984       Reagan
1984      Mondale
1988         Bush
Name: Candidate, dtype: object
```

List → [ ] → DataFrame

Multiple Column Selection

```
elections[["Candidate"]].head(6)
```

Year	Candidate
1980	Reagan
1980	Carter
1980	Anderson
1984	Reagan
1984	Mondale
1988	Bush

Numeric Slice → [ ] → DataFrame

(Multiple) Row Selection

```
elections[0:3]
```

Year	Candidate	Party	%	Result
1980	Reagan	Republican	50.7	win
1980	Carter	Democratic	41.0	loss
1980	Anderson	Independent	6.6	loss

Note: Row Selection Requires Slicing!!

`elections[0]` will not work unless the elections data frame has a column whose name is the numeric zero

- Note: It is actually possible for columns to have names that are non-String types, e.g., numeric, datetime, etc

Poll

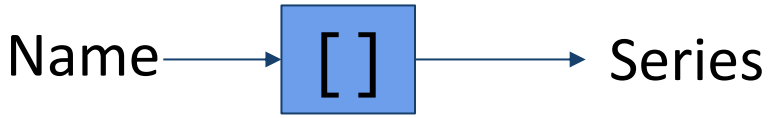
Question

```
weird = pd.DataFrame({1:["topdog","botdog"], "1":["topcat","botcat"]})
weird
```

	1	1
0	topdog	topcat
1	botdog	botcat

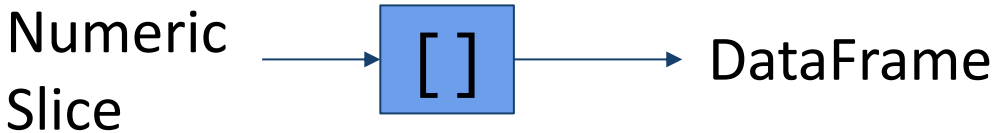
Try to predict the output of the following:

- `weird[1]`
- `weird[[“1”]]`
- `weird[1:]`



Single Column Selection

Multiple Column Selection



(Multiple) Row Selection

Let’s go to the notebook!

3-min detour (Deworming)

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

Length 23 Series where every entry is "Republican", "Democrat" or "Independent"

Length 23 Series where every entry is either "True" or "False", where "True" occurs for every independent candidate

```
elections[elections['Party'] == 'Independent']
```

	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

isin

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

Example: Suppose we want to find "Republican" or "Democratic" candidates. Could use the | operator (| means or), or we can use **isin**

- Ugly: `df[(df["Party"] == "Democratic") | (df["Party"] == "Republican")]`
- Better: `df[df["Party"].isin(["Republican", "Democratic"])]`

The Query Command

The query command provides an alternate way to combine multiple conditions

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

	Candidate	Party	%	Year	Result
0	Reagan	Republican	50.7	1980	win
3	Reagan	Republican	58.8	1984	win
5	Bush	Republican	53.4	1988	win
7	Clinton	Democratic	43.0	1992	win
10	Clinton	Democratic	49.2	1996	win

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:

- loc: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.loc.html
- iloc: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.iloc.html
- 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 of the first 3 Candidate names only for candidates that won with more than 50% of the vote

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

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

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

Question

Which of the following pandas statements returns a DataFrame of the first 3 Candidate names only for candidates that won with more than 50% of the vote.

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

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

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

```
winners = elections.query("Result == 'win'")["%"]
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

Thanks!