DSC 201: Data Analysis & Visualization

Reading Data

Dr. David Koop
Data Frame

- A dictionary of Series (labels for each series)
- A spreadsheet with column headers
- Has an index shared with each series
- Allows easy reference to any cell
- `df = DataFrame({'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada'],
    'pop': [1.5, 1.7, 3.6, 2.4]})`

- Index is automatically assigned just as with a series but can be passed in as well via index kwarg
- Can reassign column names by passing columns kwarg
DataFrame Access and Manipulation

- `df.values` → 2D NumPy array

- Accessing a column:
  - `df["<column>"]`
  - `df.<column>`

  - Both return Series
  - Dot syntax only works when the column is a valid identifier

- Assigning to a column:
  - `df["<column>"] = <scalar>` # all cells set to same value
  - `df["<column>"] = <array>` # values set in order
  - `df["<column>"] = <series>` # values set according to match

    # between df and series indexes
Dropping entries

• Can drop one or more entries

• Series:
  - `new_obj = obj.drop('c')`
  - `new_obj = obj.drop(['d', 'c'])`

• Data Frames:
  - `axis` keyword defines which axis to drop (default 0)
    - `axis==0` → rows, `axis==1` → columns
    - `axis = 'columns'`
Indexing

- Same as with NumPy arrays but can use Series's index labels

- Slicing with labels: NumPy is **exclusive**, Pandas is **inclusive**!
  
  - \( s = \text{Series}(\text{np.arange}(4)) \)
  
    \[ s[0:2] \# \text{gives two values like numpy} \]
  
  - \( s = \text{Series}(\text{np.arange}(4), \text{index}=\text{'a', 'b', 'c', 'd'}) \)
  
    \[ s[\text{'a':\text{'c'}}] \# \text{gives three values, not two!} \]

- Obtaining data subsets
  
  - []: get columns by label
  
  - loc: get rows/cols by label
  
  - iloc: get rows/cols by position (integer index)
  
  - For single cells (scalars), also have at and iat
Indexing Quiz

• \( s = \text{pd.Series(np.arange(4.), index=[4,3,2,1])} \)

• What are:
  - \( s[3] \)
  - \( s.loc[3] \)
  - \( s.iloc[3] \)

• \( s2 = \text{pd.Series(np.arange(4), index=['a','b','c','d'])} \)
  - What is \( s2[3] \)?
Indexing Quiz

• \( s = \text{pd.Series}(\text{np.arange}(4), \text{index}=[4,3,2,1]) \)

• What are:
  - \( s[3] \rightarrow 1 \)
  - \( s.loc[3] \rightarrow 1 \)
  - \( s.iloc[3] \rightarrow 3 \)

• \( s2 = \text{pd.Series}(\text{np.arange}(4), \text{index}=['a','b','c','d']) \)

• \( s2[3] \rightarrow 3 \)

• Note the ambiguity in how brackets are handled
  - index has integers, \([\] \) works like \( \text{loc}[] \)
  - index doesn't have integers, \([\] \) works like \( \text{iloc}[] \)
Filtering

• Same as with numpy arrays but allows use of column-based criteria
  - `data[data < 5] = 0`
  - `data[data['three'] > 5]`
  - `data < 5` creates a boolean data frame that can be used to select specific elements
## Arithmetic between DataFrames and Series

- **Broadcasting:** e.g. apply single row operation across all rows

**Example:**

```python
In [148]: frame
Out[148]:
     b  d  e
Utah 0  1  2
Ohio 3  4  5
Texas 6  7  8
Oregon 9  10  11

In [149]: series
Out[149]:
     b  0
Ohio 3  3
Texas 6  6
Oregon 9  9

In [150]: frame - series
Out[150]:
     b  d  e
Utah 0  1  2
Ohio 0  3  3
Texas 0  6  6
Oregon 0  9  9
```

- To broadcast over **columns**, use methods (.add, ...)

```python
In [154]: frame
Out[154]:
     b  d  e
Utah 0  1  2
Ohio 3  4  5
Texas 6  7  8
Oregon 9  10  11

In [155]: series3
Out[155]:
     Name: Utah, dtype: float64

In [156]: frame.sub(series3, axis=0)
Out[156]:
     b  d  e
Utah 0  1  2
Ohio 0  3  3
Texas 0  6  6
Oregon 0  9  9
```
Assignment 3

- Link
- Hurricane data, take 3
- Redo parts of A1 & A2 using pandas
- Part 5 shows how pandas can connect to altair
- May need to read ahead, but have tried to point to specific documentation for most of the concepts
- Due Thursday, Nov. 1
Sorting by Index (sort_index)

- **Sort by index (lexicographical):**
  
  ```python
  In [168]: obj = Series(range(4), index=['d', 'a', 'b', 'c'])
  
  In [169]: obj.sort_index()
  Out[169]:
   a    1
   b    2
   c    3
   d    0
  dtype: int64
  ```

- **DataFrame sorting:**
  
  ```python
  In [170]: frame = DataFrame(np.arange(8).reshape((2, 4)), index=['three', 'one'],
                             columns=['d', 'a', 'b', 'c'])
  
  In [171]: frame.sort_index()       
  In [172]: frame.sort_index(axis=1)
  Out[171]:                          Out[172]:
   d  a  b  c                   a  b  c  d
  one    4  5  6  7               three  1  2  3  0
  three  0  1  2  3               one    5  6  7  4
  ```

- **axis controls sort rows (0) vs. sort columns (1)**
Sorting by Value (sort_values)

• `sort_values` method on series
  - `obj.sort_values()`

• Missing values (NaN) are at the end by default (na_position controls, can be first)

• `sort_values` on DataFrame:
  - `df.sort_values(<list-of-columns>)`
  - `df.sort_values(by=['a', 'b'])`
  - Can also use `axis=1` to sort by index labels
Ranking

• `rank()` method:

In [182]: obj = Series([7, -5, 7, 4, 2, 0, 4])

In [183]: obj.rank()
Out[183]:
0  6.5
1  1.0
2  6.5
3  4.5
4  3.0
5  2.0
6  4.5
dtype: float64

• `ascending` and `method` arguments:

In [185]: obj.rank(ascending=False, method='max')
Out[185]:
0  2
1  7
2  2
3  4
4  5
5  6
6  4
dtype: float64

• Works on data frames, too

Table 5-8. Tie-breaking methods with `rank`

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>average</td>
<td>Default: assign the average rank to each entry in the equal group.</td>
</tr>
<tr>
<td>min</td>
<td>Use the minimum rank for the whole group.</td>
</tr>
<tr>
<td>max</td>
<td>Use the maximum rank for the whole group.</td>
</tr>
<tr>
<td>first</td>
<td>Assign ranks in the order the values appear in the data.</td>
</tr>
</tbody>
</table>

Axis indexes with duplicate values

Up until now all of the examples I've showed you have had unique axis labels (index values). While many pandas functions (like `reindex`) require that the labels be unique, it's not mandatory. Let's consider a small Series with duplicate indices:

In [189]: obj = Series(range(5), index=['a', 'a', 'b', 'b', 'c'])

In [190]: obj
Out[190]:
a    0
a    1
b    2
b    3

dtype: int64
Statistics

- **sum**: column sums (axis=1 gives sums over rows)
- missing values are excluded unless the whole slice is NaN
- idxmax, idxmin are like argmax, argmin (return index)
- describe: shortcut for easy stats!

```
In [204]: df.describe()
Out[204]:
       one       two
count 3.000000  2.000000
mean  3.083333 -2.900000
std   3.493685  2.262742
min   0.750000 -4.500000
25%   1.075000 -3.700000
50%   1.400000 -2.900000
75%   4.250000 -2.100000
max   7.100000 -1.300000
```

```
In [205]: obj = Series(['a', 'a', 'b', 'c'] * 4)
In [206]: obj.describe()
Out[206]:
       count     unique     top    freq
count   16     3      a       8
unique   3
         top      a  freq
         a       8
dtype: object
```
Another type of method is neither a reduction nor an accumulation. describe is one such example, producing multiple summary statistics in one shot:

In [204]: df.describe()

Out[204]:

<table>
<thead>
<tr>
<th></th>
<th>one</th>
<th>two</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>3.0000</td>
<td>2.0000</td>
</tr>
<tr>
<td>mean</td>
<td>3.0833</td>
<td>-2.9000</td>
</tr>
<tr>
<td>std</td>
<td>3.4937</td>
<td>2.2627</td>
</tr>
<tr>
<td>min</td>
<td>0.7500</td>
<td>-4.5000</td>
</tr>
<tr>
<td>25%</td>
<td>1.0750</td>
<td>-3.7000</td>
</tr>
<tr>
<td>50%</td>
<td>1.4000</td>
<td>-2.9000</td>
</tr>
<tr>
<td>75%</td>
<td>4.2500</td>
<td>-2.1000</td>
</tr>
<tr>
<td>max</td>
<td>7.1000</td>
<td>-1.3000</td>
</tr>
</tbody>
</table>

On non-numeric data, describe produces alternate summary statistics:

In [205]: obj = Series(['a', 'a', 'b', 'c'] * 4)

In [206]: obj.describe()

Out[206]:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>16</td>
</tr>
<tr>
<td>unique</td>
<td>3</td>
</tr>
<tr>
<td>top</td>
<td>a</td>
</tr>
<tr>
<td>freq</td>
<td>8</td>
</tr>
<tr>
<td>dtype</td>
<td>object</td>
</tr>
</tbody>
</table>

See Table 5-10 for a full list of summary statistics and related methods.

### Table 5-10. Descriptive and summary statistics

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>Number of non-NA values</td>
</tr>
<tr>
<td>describe</td>
<td>Compute set of summary statistics for Series or each DataFrame column</td>
</tr>
<tr>
<td>min, max</td>
<td>Compute minimum and maximum values</td>
</tr>
<tr>
<td>argmin, argmax</td>
<td>Compute index locations (integers) at which minimum or maximum value obtained, respectively</td>
</tr>
<tr>
<td>idxmin, idxmax</td>
<td>Compute index values at which minimum or maximum value obtained, respectively</td>
</tr>
<tr>
<td>quantile</td>
<td>Compute sample quantile ranging from 0 to 1</td>
</tr>
<tr>
<td>sum</td>
<td>Sum of values</td>
</tr>
<tr>
<td>mean</td>
<td>Mean of values</td>
</tr>
<tr>
<td>median</td>
<td>Arithmetic median (50% quantile) of values</td>
</tr>
<tr>
<td>mad</td>
<td>Mean absolute deviation from mean value</td>
</tr>
<tr>
<td>var</td>
<td>Sample variance of values</td>
</tr>
<tr>
<td>std</td>
<td>Sample standard deviation of values</td>
</tr>
<tr>
<td>skew</td>
<td>Sample skewness (3rd moment) of values</td>
</tr>
<tr>
<td>kurt</td>
<td>Sample kurtosis (4th moment) of values</td>
</tr>
<tr>
<td>cumsum</td>
<td>Cumulative sum of values</td>
</tr>
<tr>
<td>cummin, cummax</td>
<td>Cumulative minimum or maximum of values, respectively</td>
</tr>
<tr>
<td>cumprod</td>
<td>Cumulative product of values</td>
</tr>
<tr>
<td>diff</td>
<td>Compute 1st arithmetic difference (useful for time series)</td>
</tr>
<tr>
<td>pct_change</td>
<td>Compute percent changes</td>
</tr>
</tbody>
</table>
Unique Values and Value Counts

• unique returns an array with only the unique values (no index)
  
  - `s = Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])`
  - `s.unique() # array(['c', 'a', 'd', 'b'])`

• value_counts returns a Series with index frequencies:
  
  - `s.value_counts() # Series({'c': 3, 'a': 3, 'b': 2, 'd': 1})`
I do not claim that pandas’s NA representation is optimal, but it is simple and reasonably consistent. It’s the best solution, with good all-around performance characteristics and a simple API, that I could concoct in the absence of a true NA data type or bit pattern in NumPy’s data types. Ongoing development work in NumPy may change this in the future.

Table 5-12. NA handling methods

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dropna</td>
<td>Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate.</td>
</tr>
<tr>
<td>fillna</td>
<td>Fill in missing data with some value or using an interpolation method such as 'ffill' or 'bfill'.</td>
</tr>
<tr>
<td>isnull</td>
<td>Return like-type object containing boolean values indicating which values are missing / NA.</td>
</tr>
<tr>
<td>nonnull</td>
<td>Negation of isnull.</td>
</tr>
</tbody>
</table>

Filtering Out Missing Data

You have a number of options for filtering out missing data. While doing it by hand is always an option, dropna can be very helpful. On a Series, it returns the Series with only the non-null data and index values:

```
In [233]: from numpy import nan as NA
In [234]: data = Series([1, NA, 3.5, NA, 7])
In [235]: data.dropna()
```

```
Out[235]:
0    1.0
2    3.5
4    7.0
```

Naturally, you could have computed this yourself by boolean indexing:

```
In [236]: data[data.notnull()]
```

```
Out[236]:
0    1.0
2    3.5
4    7.0
```

With DataFrame objects, these are a bit more complex. You may want to drop rows or columns which are all NA or just those containing any NAs. dropna by default drops any row containing a missing value:

```
In [237]: data.dropna()  # default, drop rows with any missing values
```

```
  0  1.0
  2  3.5
  4  7.0
```

[W. McKinney, Python for Data Analysis]
Reading & Writing Data
Reading Data in Python

• Use the `open()` method to open a file for reading
  
  ```python
  f = open('huck-finn.txt')
  ```

• Usually, add an `'r'` as the second parameter to indicate "read"

• Can iterate through the file (think of the file as a collection of lines):
  ```python
  f = open('huck-finn.txt', 'r')
  for line in f:
    if 'Huckleberry' in line:
      print(line.strip())
  ```

• Using `line.strip()` because the read includes the newline, and print writes a newline so we would have double-spaced text

• Closing the file: `f.close()`
With Statement: Improved File Handling

• With statement allows "enter" and "exit" handling (kind of like the finally clause):

• In the previous example, we need to remember to call `f.close()`

• Using a with statement, this is done automatically:

  ```python
  with open('huck-finn.txt', 'r') as f:
      for line in f:
          if 'Huckleberry' in line:
              print(line.strip())
  ```

• This is more important for writing files!

  ```python
  with open('output.txt', 'w') as f:
      for k, v in counts.items():
          f.write(k + ':' + v + '\n')
  ```

• Without `with`, we need `f.close()`
Comma-separated values (CSV) Format

• Comma is a field separator, newlines denote records
  - a,b,c,d,message
  1,2,3,4,hello
  5,6,7,8,world
  9,10,11,12,foo

• May have a header (a,b,c,d,message), but not required

• No type information: we do not know what the columns are (numbers, strings, floating point, etc.)
  - Default: just keep everything as a string
  - Type inference: Figure out what type to make each column based on what they look like

• What about commas in a value? → double quotes
Delimiter-separated Values

• Comma is a **delimiter**, specifies boundary between fields
• Could be a tab, pipe (|), or perhaps spaces instead
• All of these follow similar styles to CSV
Fixed-width Format

- Old school

- Each field gets a certain number of spots in the file

- Example:

  - id8141    360.242940   149.910199   11950.7
  - id1594    444.953632   166.985655   11788.4
  - id1849    364.136849   183.628767   11806.2
  - id1230    413.836124   184.375703   11916.8
  - id1948    502.953953   173.237159   12468.3

- Specify exact character ranges for each field, e.g. 0-6 is the id
# Reading & Writing Data in Pandas

<table>
<thead>
<tr>
<th>Format Type</th>
<th>Data Description</th>
<th>Reader</th>
<th>Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>CSV</td>
<td>read_csv</td>
<td>to_csv</td>
</tr>
<tr>
<td>text</td>
<td>JSON</td>
<td>read_json</td>
<td>to_json</td>
</tr>
<tr>
<td>text</td>
<td>HTML</td>
<td>read_html</td>
<td>to_html</td>
</tr>
<tr>
<td>text</td>
<td>Local clipboard</td>
<td>read_clipboard</td>
<td>to_clipboard</td>
</tr>
<tr>
<td>binary</td>
<td>MS Excel</td>
<td>read_excel</td>
<td>to_excel</td>
</tr>
<tr>
<td>binary</td>
<td>HDF5 Format</td>
<td>read_hdf</td>
<td>to_hdf</td>
</tr>
<tr>
<td>binary</td>
<td>Feather Format</td>
<td>read_feather</td>
<td>to_feather</td>
</tr>
<tr>
<td>binary</td>
<td>Parquet Format</td>
<td>read_parquet</td>
<td>to_parquet</td>
</tr>
<tr>
<td>binary</td>
<td>Msgpack</td>
<td>read_msgpack</td>
<td>to_msgpack</td>
</tr>
<tr>
<td>binary</td>
<td>Stata</td>
<td>read_stata</td>
<td>to_stata</td>
</tr>
<tr>
<td>binary</td>
<td>SAS</td>
<td>read_sas</td>
<td></td>
</tr>
<tr>
<td>binary</td>
<td>Python Pickle Format</td>
<td>read_pickle</td>
<td>to_pickle</td>
</tr>
<tr>
<td>SQL</td>
<td>SQL</td>
<td>read_sql</td>
<td>to_sql</td>
</tr>
<tr>
<td>SQL</td>
<td>Google Big Query</td>
<td>read_gbq</td>
<td>to_gbq</td>
</tr>
</tbody>
</table>

[https://pandas.pydata.org/pandas-docs/stable/io.html]
Types of arguments for readers

- Indexing: choose a column to index the data, get column names from file or user
- Type inference and data conversion: automatic or user-defined
- Datetime parsing: can combine information from multiple columns
- Iterating: deal with very large files
- Unclean Data: skip rows (e.g. comments) or deal with formatted numbers (e.g. 1,000,345)
read_csv

• Convenient method to read csv files
• Lots of different options to help get data into the desired format
• Basic: df = pd.read_csv(fname)
• Parameters:
  - path: where to read the data from
  - sep (or delimiter): the delimiter (',', ' ', '	', '\s+')
  - header: if None, no header
  - index_col: which column to use as the row index
  - names: list of header names (e.g. if the file has no header)
  - skiprows: number of list of lines to skip
More read_csv/read_tables arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>skiprows</td>
<td>Number of rows at beginning of file to ignore or list of row numbers (starting from 0) to skip.</td>
</tr>
<tr>
<td>na_values</td>
<td>Sequence of values to replace with NA.</td>
</tr>
<tr>
<td>comment</td>
<td>Character(s) to split comments off the end of lines.</td>
</tr>
<tr>
<td>parse_dates</td>
<td>Attempt to parse data to datetime; False by default. If True, will attempt to parse all columns. Otherwise can specify a list of column numbers or name to parse. If element of list is tuple or list, will combine multiple columns together and parse to date (e.g., if date/time split across two columns).</td>
</tr>
<tr>
<td>keep_date_col</td>
<td>If joining columns to parse date, keep the joined columns; False by default.</td>
</tr>
<tr>
<td>converters</td>
<td>Dict containing column number of name mapping to functions (e.g., {'foo': f} would apply the function f to all values in the 'foo' column).</td>
</tr>
<tr>
<td>dayfirst</td>
<td>When parsing potentially ambiguous dates, treat as international format (e.g., 7/6/2012 -&gt; June 7, 2012); False by default.</td>
</tr>
<tr>
<td>date_parser</td>
<td>Function to use to parse dates.</td>
</tr>
<tr>
<td>nrows</td>
<td>Number of rows to read from beginning of file.</td>
</tr>
<tr>
<td>iterator</td>
<td>Return a TextParser object for reading file piecemeal.</td>
</tr>
<tr>
<td>chunksize</td>
<td>For iteration, size of file chunks.</td>
</tr>
<tr>
<td>skip_footer</td>
<td>Number of lines to ignore at end of file.</td>
</tr>
<tr>
<td>verbose</td>
<td>Print various parser output information, like the number of missing values placed in non-numeric columns.</td>
</tr>
<tr>
<td>encoding</td>
<td>Text encoding for Unicode (e.g., 'utf-8' for UTF-8 encoded text).</td>
</tr>
<tr>
<td>squeeze</td>
<td>If the parsed data only contains one column, return a Series.</td>
</tr>
<tr>
<td>thousands</td>
<td>Separator for thousands (e.g., ',', ' or '.').</td>
</tr>
</tbody>
</table>
Chunked Reads

- With very large files, we may not want to read the entire file

- Why?
  - Time
  - Want to understand part of data before processing all of it

- Reading only a few rows:
  - `df = pd.read_csv('example.csv', nrows=5)`

- Reading chunks:
  - Get an iterator that returns the next chunk of the file
  - `chunker = pd.read_csv('example.csv', chunksize=1000)`
  - `for piece in chunker:`
    - `process_data(piece)`
Python csv module

- Also, can read csv files outside of pandas using csv module

  ```python
  import csv
  with open('persons_of_concern.csv', 'r') as f:
      for i in range(3):
          next(f)
  reader = csv.reader(f)
  records = [r for r in reader] # r is a list
  ```

- or

  ```python
  import csv
  with open('persons_of_concern.csv', 'r') as f:
      for i in range(3):
          next(f)
  reader = csv.DictReader(f)
  records = [r for r in reader] # r is a dict
  ```
Writing CSV data with pandas

• **Basic**: `df.to_csv(<fname>)`

• Change delimiter with `sep` kwarg:
  - `df.to_csv('example.dsv', sep='|')`

• Change missing value representation
  - `df.to_csv('example.dsv', na_rep='NULL')`

• Don't write row or column labels:
  - `df.to_csv('example.csv', index=False, header=False)`

• Series may also be written to csv
JavaScript Object Notation (JSON)

- A format for web data
- Looks very similar to python dictionaries and lists
- Example:

  ```json
  {"name": "Wes",
   "places_lived": ["United States", "Spain", "Germany"],
   "pet": null,
   "siblings": [{"name": "Scott", "age": 25, "pet": "Zuko"},
   {"name": "Katie", "age": 33, "pet": "Cisco"}]
  }
  ```

- Only contains literals (no variables) but allows null
- Values: strings, arrays, dictionaries, numbers, booleans, or null
  - Dictionary keys must be strings
  - Quotation marks help differentiate string or numeric values
eXtensible Markup Language (XML)

- Older, self-describing format with nesting
- Each field has tags
- Example:

  ```xml
  <INDICATOR>
    <INDICATOR_SEQ>373889</INDICATOR_SEQ>
    <PARENT_SEQ></PARENT_SEQ>
    <AGENCY_NAME>Metro-North Railroad</AGENCY_NAME>
    <INDICATOR_NAME>Escalator Avail.</INDICATOR_NAME>
    <PERIOD_YEAR>2011</PERIOD_YEAR>
    <PERIOD_MONTH>12</PERIOD_MONTH>
    <CATEGORY>Service Indicators</CATEGORY>
    <FREQUENCY>M</FREQUENCY>
    <YTD_TARGET>97.00</YTD_TARGET>
  </INDICATOR>
  
  Top element is the root
What is the problem with reading this data?

- [{"name": "Wes",  "places_lived": ["United States", "Spain", "Germany"],  "pet": null,  "siblings": [    {"name": "Scott", "age": 25, "pet": "Zuko"},    {"name": "Katie", "age": 33, "pet": "Cisco"}  ]}, {"name": "Nia",  "address": {"street": "143 Main",  "city": "New York",  "state": "New York"},  "pet": "Fido",  "siblings": [    {"name": "Jacques", "age": 15, "pet": "Fido"}  ]}, ...]
Reading JSON data

• Python has a built-in `json` module
  - with `open('example.json')` as f:
    data = json.load(f)
  - Can also load/dump to strings:
    • `json.loads, json.dumps`

• Pandas has `read_json, to_json` methods
JSON Orientation

- Indication of expected JSON string format. Compatible JSON strings can be produced by `to_json()` with a corresponding orient value. The set of possible orients is:

  - **split**: dict like `{index -> [index],
    columns -> [columns],
    data -> [values]}`

  - **records**: list like
    `[{column -> value}, ... , {column -> value}]`

  - **index**: dict like `{index -> {column -> value}}`

  - **columns**: dict like `{column -> {index -> value}}`

  - **values**: just the values array