On TAP:

Module 2: Data Frames

Kyle H. Ambert, PhD

Intel Big Data Solutions, Datacenter Group

- See which frames are available to us (get_frame_names)
- Load a frame into current namespace
- Examine frame methods
- Frame debugging
- add_columns


Data

TAP Data Frames

TAP Data Frames

- A meaningful collection of columns of different data types

TAP Data Frames

- A meaningful collection of columns of different data types
- Analogous to Pandas Frames

TAP Data Frames

- A meaningful collection of columns of different data types
- Analogous to Pandas Frames

TAP Data Frames

- A meaningful collection of columns of different data types
- Analogous to Pandas Frames

- Columns are strongly-typed

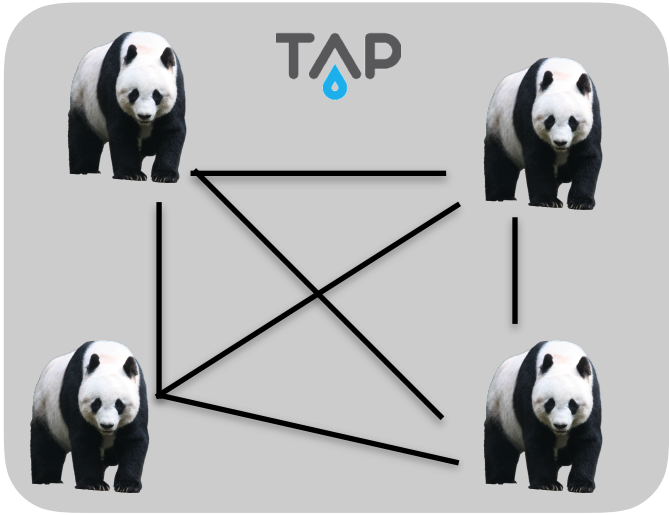
TAP Data Frames

- A meaningful collection of columns of different data types
- Analogous to Pandas Frames

- Columns are strongly-typed
- Within a row, elements can be accessed like python attributes, or like entries in a dictionary

TAP Data Frames

- A meaningful collection of columns of different data types
- Analogous to Pandas Frames
    - only distributed!
- Columns are strongly-typed
- Within a row, elements can be accessed like python attributes, or like entries in a dictionary



Data Types

All data manipulated and stored using frames and graphs must fit into one of the supported Python data types.

```
>>> ta.valid_data_types

float32, float64, ignore, int32, int64, unicode, vector(n), datetime
(and aliases: float->float64, int->int32, list->vector, long->int64, str->unicode)
```

datetime	[ALPHA] object for date and time; equivalent to python's datetime.datetime class. Converts to and from strings using the ISO 8601 format. Inside the server, the object is represented by the nscala/joda DateTime object. When interfacing with various data sources and sinks that use different data types for datetime, the datetime value will be converted to a string by default.
float32	32-bit floating point number; equivalent to numpy.float32
float64	64-bit floating point number; equivalent to numpy.float64
ignore	type available to describe a field in a data source that the parser should ignore
int32	32-bit integer; equivalent to numpy.int32
int64	32-bit integer; equivalent to numpy.int64
unicode	Python's unicode representation for strings.
vector(n)	[ALPHA] Ordered list of n float64 numbers (array of fixed-length n); uses numpy.ndarray

Note: Numpy values of positive infinity (np.inf), negative infinity (-np.inf) or nan (np.nan) are treated as Python's None when sent to the server. Results of any user-defined functions which deal with such values are automatically converted to None. Any further usage of those data points should treat the values as None.

Data Types

All data manipulated and stored using frames and graphs must fit into one of the supported Python data types.

```
>>> ta.valid_data_types

float32, float64, ignore, int32, int64, unicode, vector(n), datetime
(and aliases: float->float64, int->int32, list->vector, long->int64, str->unicode)
```

datetime	[ALPHA] object for date and time; equivalent to python's datetime.datetime class. Converts to and from strings using the ISO 8601 format. Inside the server, the object is represented by the nscala/joda DateTime object. When interfacing with various data sources and sinks that use different data types for datetime, the datetime value will be converted to a string by default.
float32	32-bit floating point number; equivalent to numpy.float32
float64	64-bit floating point number; equivalent to numpy.float64
ignore	type available to describe a field in a data source that the parser should ignore
int32	32-bit integer; equivalent to numpy.int32
int64	32-bit integer; equivalent to numpy.int64
unicode	Python's unicode representation for strings.
vector(n)	[ALPHA] Ordered list of n float64 numbers (array of fixed-length n); uses numpy.ndarray

Note: Numpy values of positive infinity (np.inf), negative infinity (-np.inf) or nan (np.nan) are treated as Python's None when sent to the server. Results of any user-defined functions which deal with such values are automatically converted to None. Any further usage of those data points should treat the values as None.

Additional data types are in development!


Data


Data

!

— add_columns!

- Arguably, the main workhorse function of data wrangling
- We'll explore this extensively in coding sessions


```
Frame.add_columns(self, func, schema, columns_accessed=None)
```

— add_columns!

- Arguably, the main workhorse function of data wrangling
- We'll explore this extensively in coding sessions

— add_columns!

- Arguably, the main workhorse function of data wrangling
- We'll explore this extensively in coding sessions

```
Frame.add_columns(self, func, schema, columns_accessed=None)

>>> frame.inspect()
        [#]  name      age  tenure  phone
        ===================================
        [0]  Fred       39      16  555-1234
        [1]  Susan      33       3  555-0202
        [2]  Thurston   65      26  555-4510
        [3]  Judy       44      14  555-2183

>>> frame.add_columns(lambda row: row.age - 18,
('adult_years', ta.int32))
>>> frame.inspect()
        [#]  name      age  tenure  phone     adult_years
        ============================================================
        [0]  Fred       39      16  555-1234            21
        [1]  Susan      33       3  555-0202            15
        [2]  Thurston   65      26  555-4510            47
        [3]  Judy       44      14  555-2183            26
```

— add_columns!

- Arguably, the main workhorse function of data wrangling
- We'll explore this extensively in coding sessions

```
Frame.add_columns(self, func, schema, columns_accessed=None)

>>> frame.inspect()
        [#]  name      age  tenure  phone
        ===================================
        [0]  Fred       39      16  555-1234
        [1]  Susan      33       3  555-0202
        [2]  Thurston   65      26  555-4510
        [3]  Judy       44      14  555-2183

>>> frame.add_columns(lambda row: row.age - 18,
('adult_years', ta.int32))
>>> frame.inspect()
        [#]  name      age  tenure  phone     adult_years
        ============================================================
        [0]  Fred       39      16  555-1234            21
        [1]  Susan      33       3  555-0202            15
        [2]  Thurston   65      26  555-4510            47
        [3]  Judy       44      14  555-2183            26
```

You can use add_columns to generate multiple columns simultaneously!

Data

¡Pro Tip!

add_columns can be used intuitively when wrapped as a function!

```
def add_offset_age(row):
    my_json = json.loads(row[0])
    AGE = my_json['AGE'] if 'AGE' in my_json else 0.0
    rn = random.randint(a=0, b=1000)
    return AGE + rn

tutorial_inpat.add_columns(add_offset_age, ("OFFSET_AGE", ia.float64))
```

add_columns

- Arguably
function

- We'll explore this extensively in coding sessions

```
Frame.add_columns(self, func, schema, columns_accessed=None)
```

```
 age  tenure  phone
=======================
  39      16  555-1234
  33       3  555-0202
  65      26  555-4510
  44      14  555-2183
```

```
>>> frame.add_columns(lambda row: row.age - 18,
('adult_years', ta.int32))
>>> frame.inspect()
      [#]  name      age  tenure  phone     adult_years
     ==============================================
     [0]  Fred       39      16  555-1234           21
     [1]  Susan      33       3  555-0202           15
     [2]  Thurston   65      26  555-4510           47
     [3]  Judy       44      14  555-2183           26
```

You can use add_columns to generate multiple columns simultaneously!

Inspect

Inspect

```
inspect(self, n=10,
         offset=0,
         columns=None,
         wrap='inspect_settings',
         truncate='inspect_settings',
         round='inspect_settings',
         width='inspect_settings',
         margin='inspect_settings',
         with_types='inspect_settings'
        )
```

Inspect

```
inspect(self, n=10,
        offset=0,
        columns=None,
        wrap='inspect_settings',
        truncate='inspect_settings',
        round='inspect_settings',
        width='inspect_settings',
        margin='inspect_settings',
        with_types='inspect_settings'
       )
```

```
>>> frame.inspect(4)
      [#]  animal    name    age  weight
      ================================
      [0]  human     George    8   542.5
      [1]  human     Ursula    6   495.0
      [2]  ape       Ape      41   400.0
      [3]  elephant  Shep      5  8630.0
```

Inspect

```
inspect(self, n=10,
        offset=0,
        columns=None,
        wrap='inspect_settings',
        truncate='inspect_settings',
        round='inspect_settings',
        width='inspect_settings',
        margin='inspect_settings',
        with_types='inspect_settings'
       )
```

```
>>> frame.inspect(4)
      [#]  animal    name    age  weight
      ================================
      [0]  human     George    8   542.5
      [1]  human     Ursula    6   495.0
      [2]  ape       Ape      41   400.0
      [3]  elephant  Shep      5  8630.0
```

Inspect prints to std.out!

Inspect

row_count, column_names

Inspect

row_count, column_names

row_count → returns an integer

column_names → returns a list

Inspect

row_count, column_names

row_count → returns an integer

column_names → returns a list

row_count and column_names are attributes of a Frame object!

Inspect

row_count, column_names

sort


```
Frame.sort(self, columns, ascending=True)
```

Inspect

row_count, column_names

sort

Inspect

row_count, column_names

sort

```
Frame.sort(self, columns, ascending=True)
```

```
>>> frame.inspect()
        [#]  col1  col2
        ==================
        [0]     3  foxtrot
        [1]     1  charlie
        [2]     3  bravo
        [3]     2  echo
        [4]     4  delta
        [5]     3  alpha

>>> frame.sort('col1')
>>> frame.inspect()
        [#]  col1  col2
        ==================
        [0]     1  charlie
        [1]     2  echo
        [2]     3  foxtrot
        [3]     3  bravo
        [4]     3  alpha
        [5]     4  delta
```

Inspect

row_count, column_names

sort

```
Frame.sort(self, columns, ascending=True)
```

```
>>> frame.inspect()
        [#]  col1  col2
        ==================
        [0]     3  foxtrot
        [1]     1  charlie
        [2]     3  bravo
        [3]     2  echo
        [4]     4  delta
        [5]     3  alpha

>>> frame.sort('col1')
>>> frame.inspect()
        [#]  col1  col2
        ==================
        [0]     1  charlie
        [1]     2  echo
        [2]     3  foxtrot
        [3]     3  bravo
        [4]     3  alpha
        [5]     4  delta
```

Sort modifies a frame in place!

Inspect
row_count, column_names
sort
filter


```
Frame.filter(self, predicate)
```

Inspect
row_count, column_names
sort
filter

Inspect
row_count, column_names
sort
filter

Frame.filter(self, predicate)

```
>> frame.inspect()
        [#]  name      age  tenure  phone
        ===================================
        [0]  Fred       39      16  555-1234
        [1]  Susan      33       3  555-0202
        [2]  Thurston   65      26  555-4510
        [3]  Judy       44      14  555-2183
>>> frame.filter(lambda row: row.tenure >= 15)
>>> frame.inspect()
        [#]  name      age  tenure  phone
        ===================================
        [0]  Fred       39      16  555-1234
        [1]  Thurston   65      26  555-4510
```

Inspect
row_count, column_names
sort
filter

Frame.filter(self, predicate)

```
>> frame.inspect()
        [#]  name      age  tenure  phone
        ===================================
        [0]  Fred       39      16  555-1234
        [1]  Susan      33       3  555-0202
        [2]  Thurston   65      26  555-4510
        [3]  Judy       44      14  555-2183
>>> frame.filter(lambda row: row.tenure >= 15)
>>> frame.inspect()
        [#]  name      age  tenure  phone
        ===================================
        [0]  Fred       39      16  555-1234
        [1]  Thurston   65      26  555-4510
```

`predicate` evaluates rows to boolean; '`False`' rows are dropped from the frame
`filter` modifies the frame in place!

Inspect
row_count, column_names
sort
filter
group_by

Inspect
row_count, column_names
sort
filter
group_by

Additional data types are in development!

Inspect
row_count, column_names
sort
filter
group_by

```
>>> frame.inspect()
        [#]  a  b        c     d       e  f    g
        ======================================
        [0]  1  alpha     3.0  small   1  3.0  9
        [1]  1  bravo     5.0  medium  1  4.0  9
        [2]  1  alpha     5.0  large   1  8.0  8
        [3]  2  bravo     8.0  large   1  5.0  7
        [4]  2  charlie  12.0  medium  1  6.0  6
        [5]  2  bravo     7.0  small   1  8.0  5
        [6]  2  bravo    12.0  large   1  6.0  4

>>> b_count = frame.group_by('b', ta.agg.count)
>>> b_count.inspect()
        [#]  b        count
        ==================
        [0]  alpha        2
        [1]  bravo        4
        [2]  charlie      1
```

Additional data types are in development!

Inspect
row_count, column_names
sort
filter
group_by

```
Frame.group_by(self, group_by_columns, *aggregation_arguments)
>>> frame.inspect()
        [#]  a  b        c     d       e  f    g
        ======================================
        [0]  1  alpha     3.0  small   1  3.0  9
        [1]  1  bravo     5.0  medium  1  4.0  9
        [2]  1  alpha     5.0  large   1  8.0  8
        [3]  2  bravo     8.0  large   1  5.0  7
        [4]  2  charlie  12.0  medium  1  6.0  6
        [5]  2  bravo     7.0  small   1  8.0  5
        [6]  2  bravo    12.0  large   1  6.0  4

>>> b_count = frame.group_by('b', ta.agg.count)
>>> b_count.inspect()
        [#]  b        count
        ==================
        [0]  alpha        2
        [1]  bravo        4
        [2]  charlie      1
```

Additional data types are in development!

code!

Problems:

[**1**] Using a copy of the `inpat` data set, identify the first five data fields in the *json* of the first row in the frame.

[**2**] What is the data type of the Using a copy of the `inpat` data set, identify the first five data fields in the *json* of the first row in the frame.

[**3**] What is the maximum value of the field '`DRG_WEIGHT`' in the data set?

[**4**] In the same data set, how many '`DRG_WEIGHT`' values are between 0 and 5? Between 5 and 10? Between 10 and 15?