Data Science on TAP

Kyle H. Ambert, PhD

November 18th, 2015

Intel Big Data Solutions, Datacenter Group

Data Science on TAP

TAP:

- Provides the infrastructure and plumbing for deploying analytics applications to production

Analytics Toolkit:

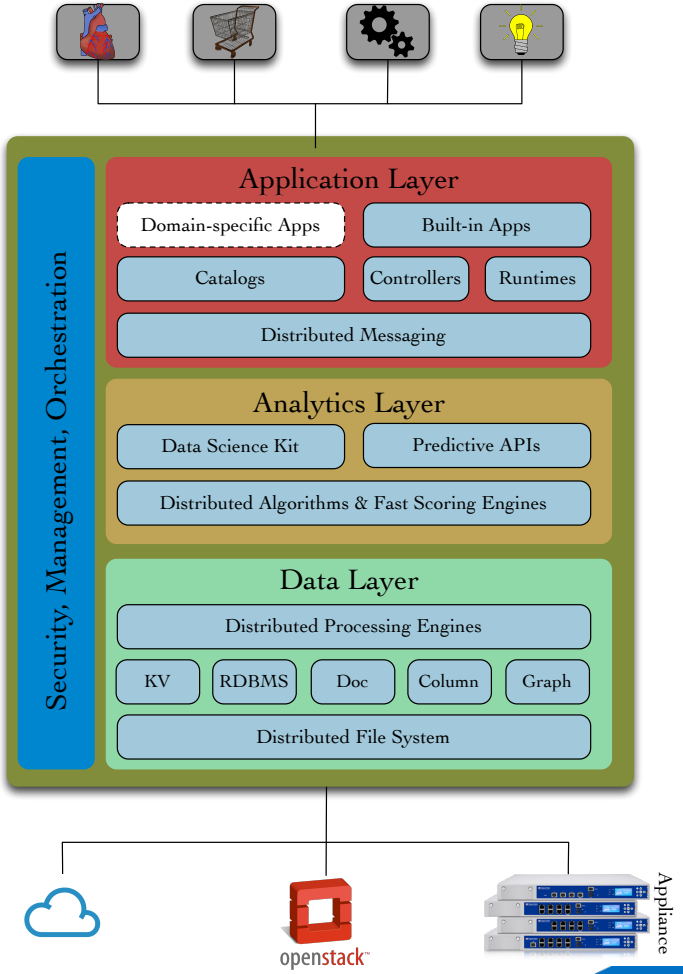
- Provides the analytics functionality
- Allows for iterative application development and experimentation

Taken together:

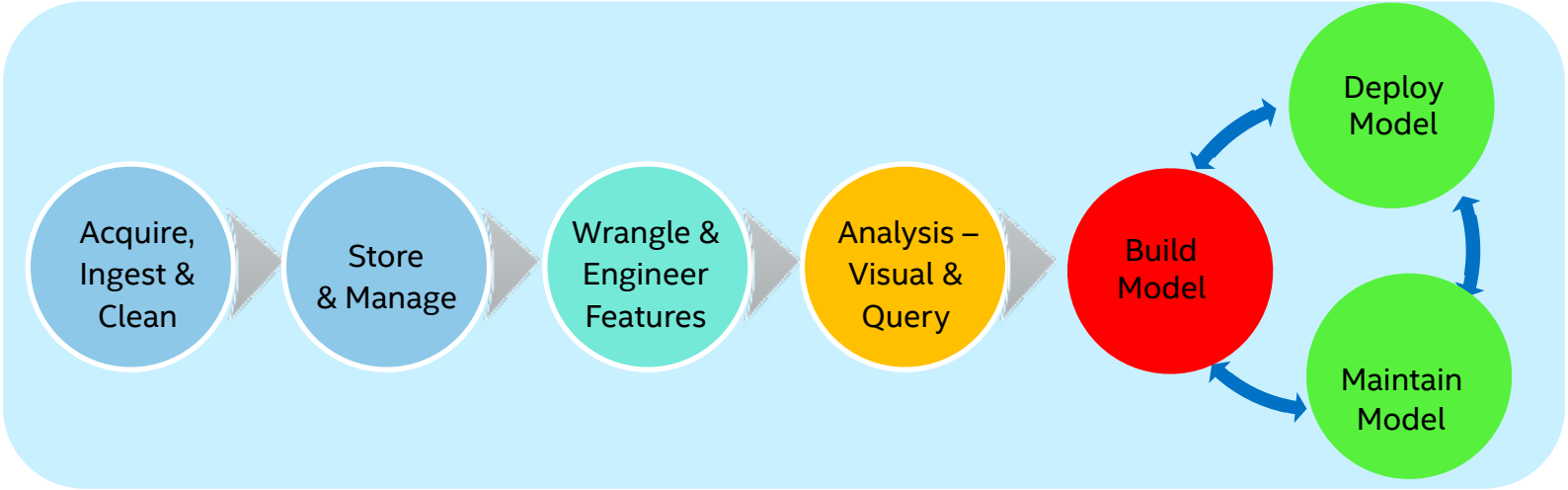
- Less moving between applications and/or languages
- Efficient use of analyst time, minimization of human error

Trusted Analytics Platform (TAP)

Open source software project that accelerates creation of Cloud-native apps driven by big data analytics. TAP makes it easier for developers to collaborate with data scientists by providing a shared environment for advanced analytics on big data.



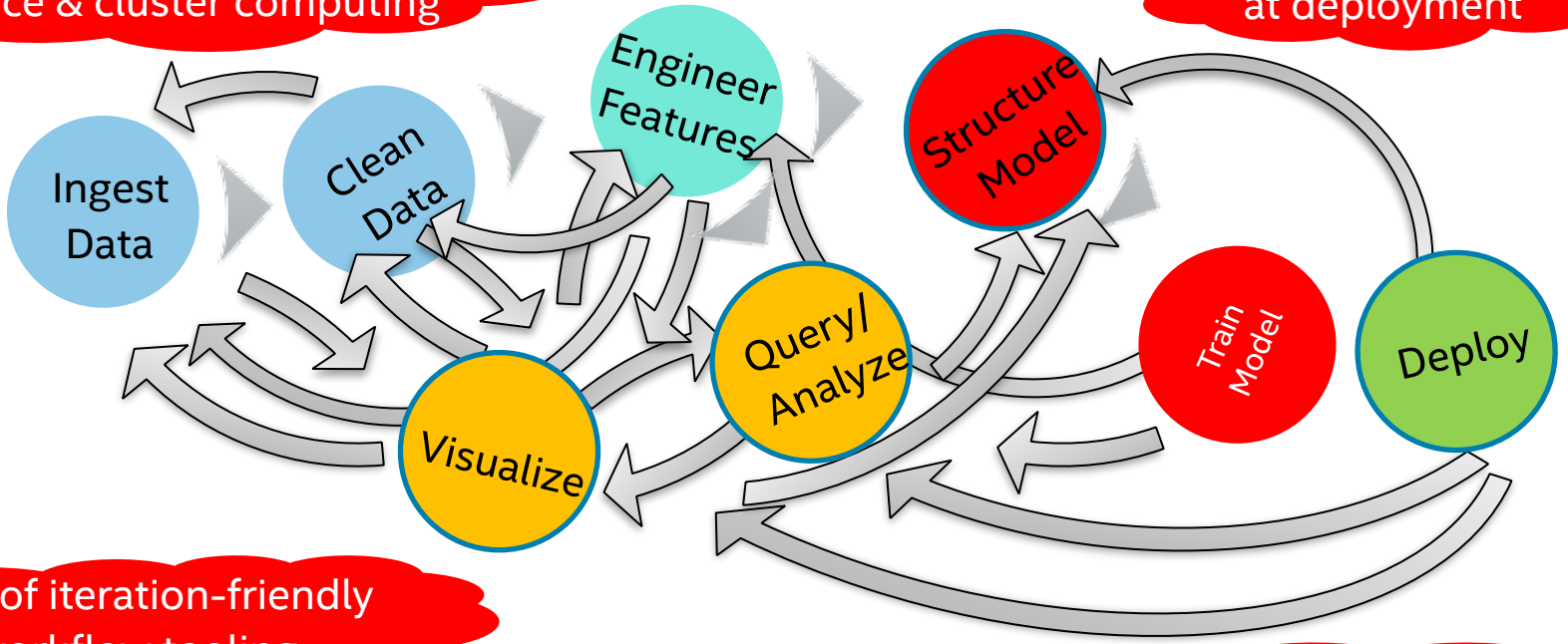
The Data Science Workflow



Challenges Across The Workflow

Skill shortages: data science & cluster computing

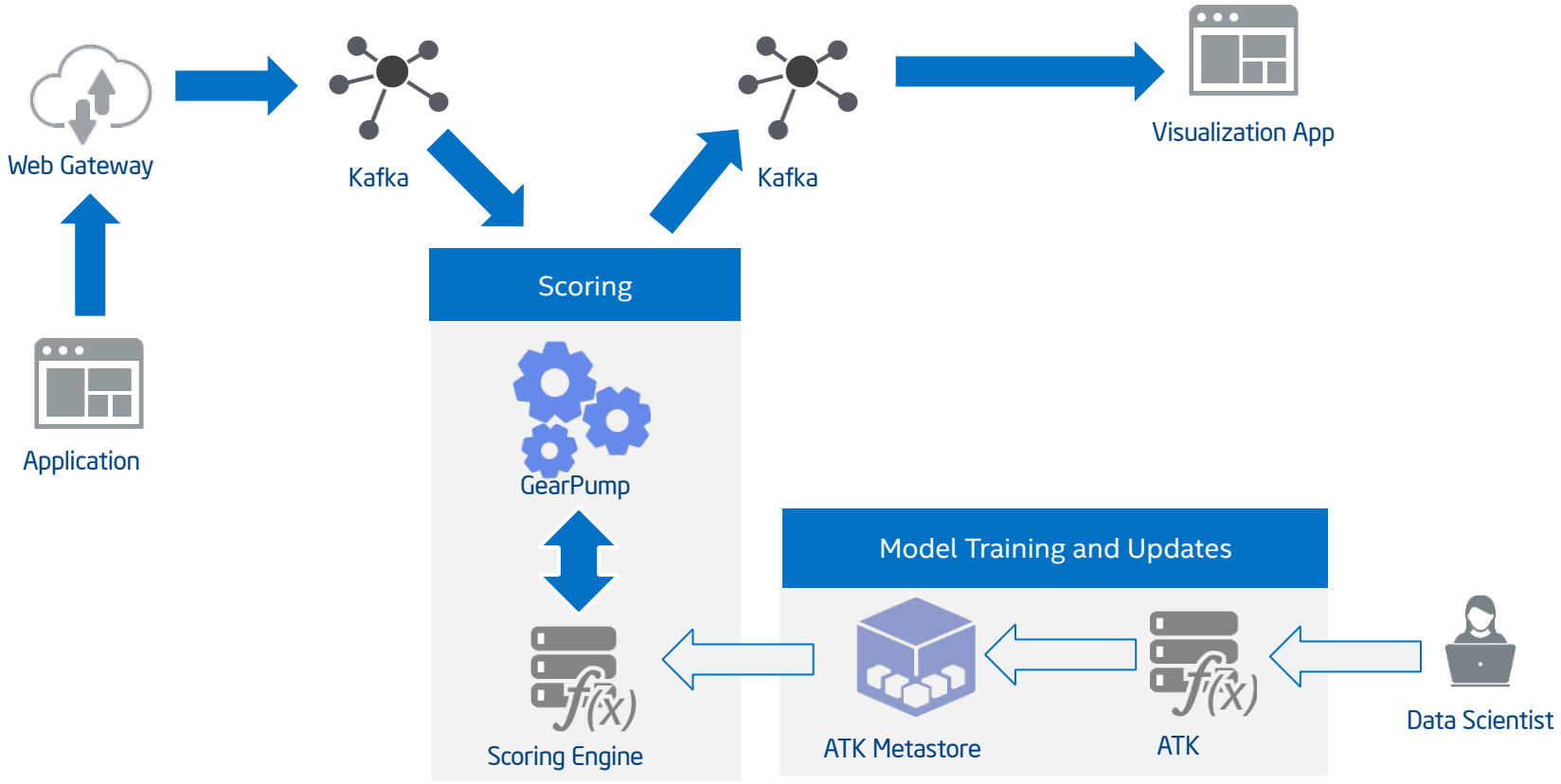
Process starts over at deployment



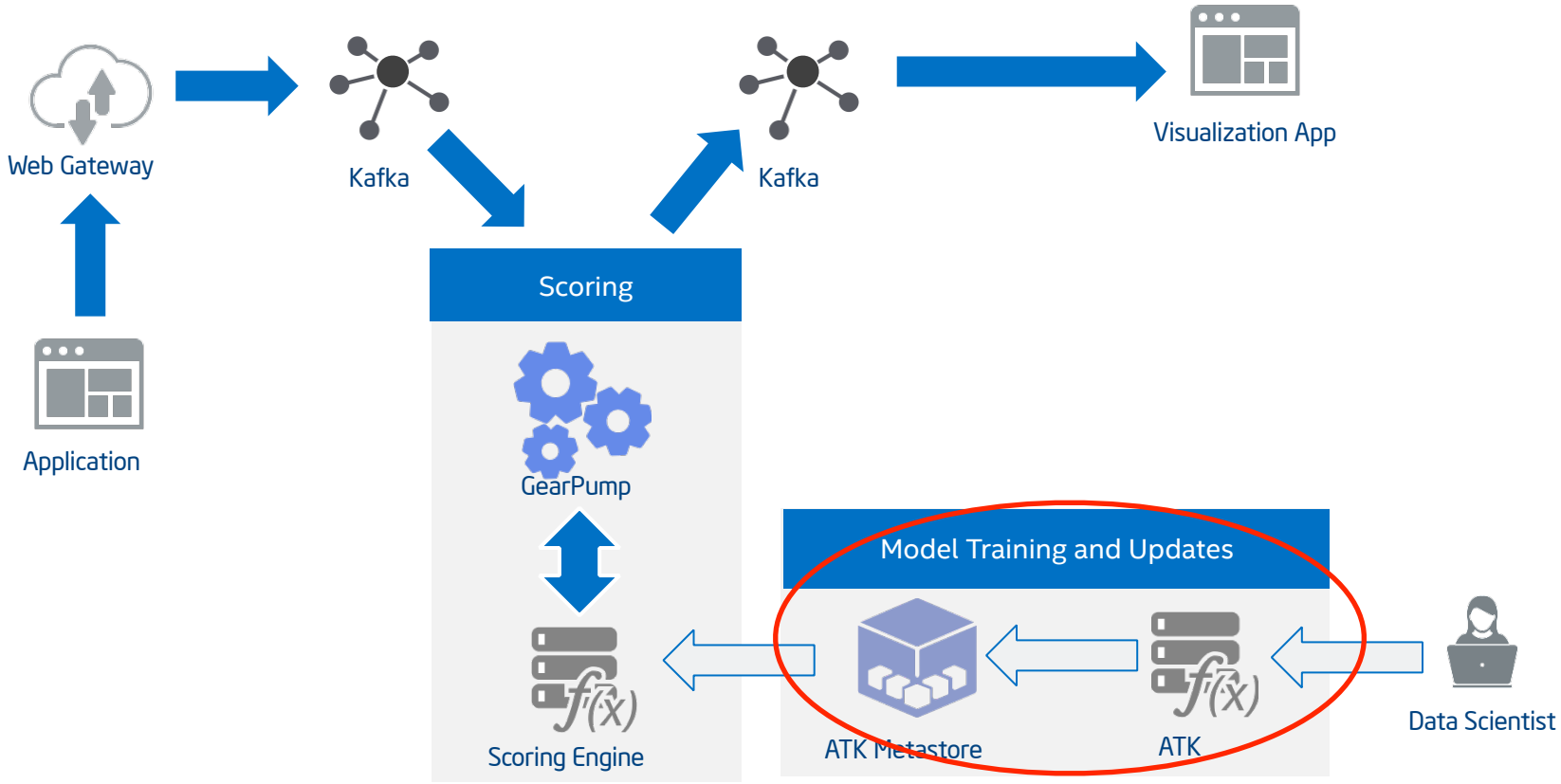
Lack of iteration-friendly workflow tooling

Limited methods to explore hidden patterns in big data

Example: App With End to End Scoring



Example: App With End to End Scoring



Data Science on TAP
Data Ingest, Data Frames, & Model Building

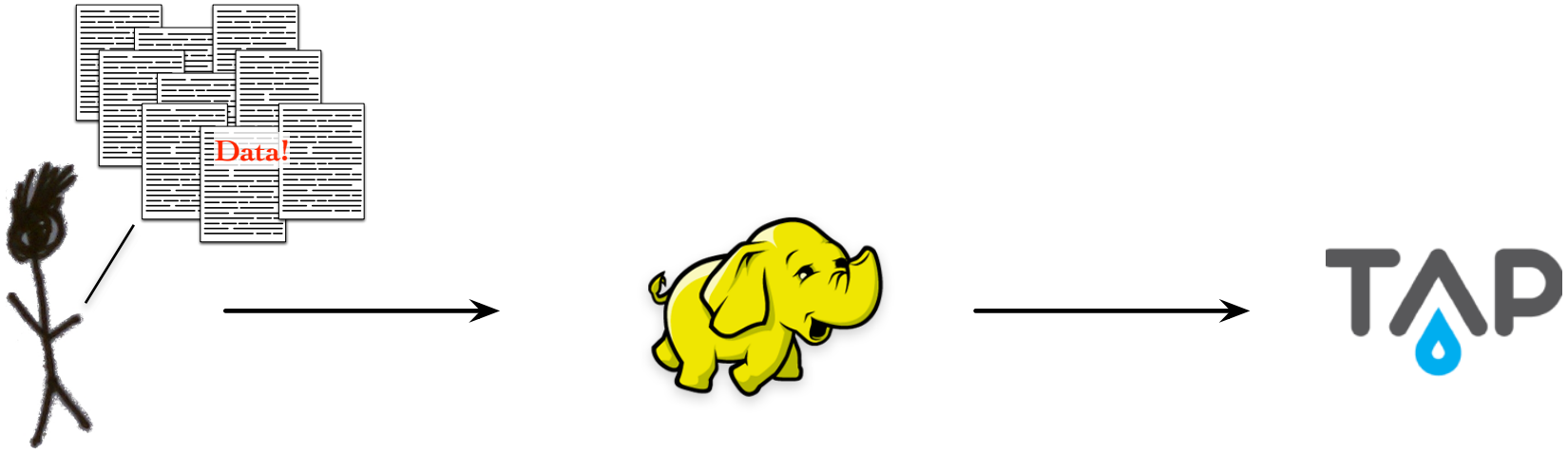
Kyle H. Ambert, PhD

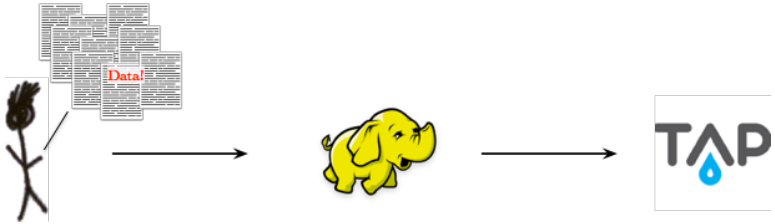
Intel Big Data Solutions, Datacenter Group

Ingest:

- Getting data into TAP
- Getting data into atk
- Moving from raw to the beginning of structure
- File types:
  - csv
  - xml
  - json

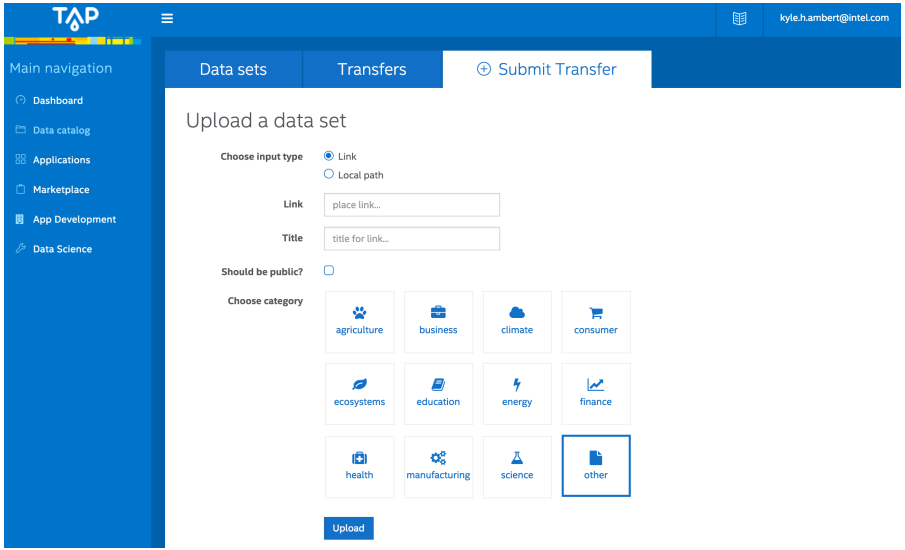
Data on TAP





Getting data into the Toolkit

- Use the console front-end
    - Data Catalog page
- Submit transfer from local path (or remote url)
- Upload!
- Data can be previewed from the Data Sets tab



Input Data Manually

[**1**] Define the **Data**

- Iterative error-prone drudgery
- One-off, ad hoc models in isolation

[**2**] Define the **Schema**

- A list of tuples: `('column_name', data_type)`

[**3**] Use the `ia.UploadRows` function

Input Data Manually

[**1**] Define the **Data**

- Iterative error-prone drudgery
- One-off, ad hoc models in isolation

[**2**] Define the **Schema**

- A list of tuples: `('column_name', data_type)`

[**3**] Use the `ia.UploadRows` function

Input Data Manually

[**1**] Define the **Data**

- Iterative error-prone drudgery
- One-off, ad hoc models in isolation

[**2**] Define the **Schema**

- A list of tuples: `('column_name', data_type)`

[**3**] Use the `ia.UploadRows` function

Example:

```
animals = ia.Frame(ia.UploadRows([['puppy', 'Harrison'], ['cat', 'wahlberg']], [('animal', str), ('name', str)]))
```

Input Data Manually

[**1**] Define the **Data**

- Iterative error-prone drudgery
- One-off, ad hoc models in isolation

[**2**] Define the **Schema**

- A list of tuples: `('column_name', data_type)`

[**3**] Use the `ia.UploadRows` function

Example:

```
animals = ia.Frame(ia.UploadRows([['puppy', 'Harrison'], ['cat', 'wahlberg']], [('animal', str), ('name', str)]))
```

All columns must be the same length! All data types must be ***serializable***!

Character-separated Value Files (*.csv*)

[**1**] Define the schema

- Iterative error-prone drudgery
- One-off, ad hoc models in isolation

[**2**] ta.CsvFile

- Single-threaded, single-node processing

```
__init__(self, file_name, schema, delimiter=',', skip_header_lines=0)
```

Example:

```
import trustedanalytics as ta
ta.CsvFile("raw_data.csv", schema=[("col1", ta.int32), ("col2", ta.float32)])
```

Character-separated Value Files (*.csv*)

[**1**] Define the schema

- Iterative error-prone drudgery
- One-off, ad hoc models in isolation

[**2**] ta.CsvFile

- Single-threaded, single-node processing

```
__init__(self, file_name, schema, delimiter=',', skip_header_lines=0)
```

Example:

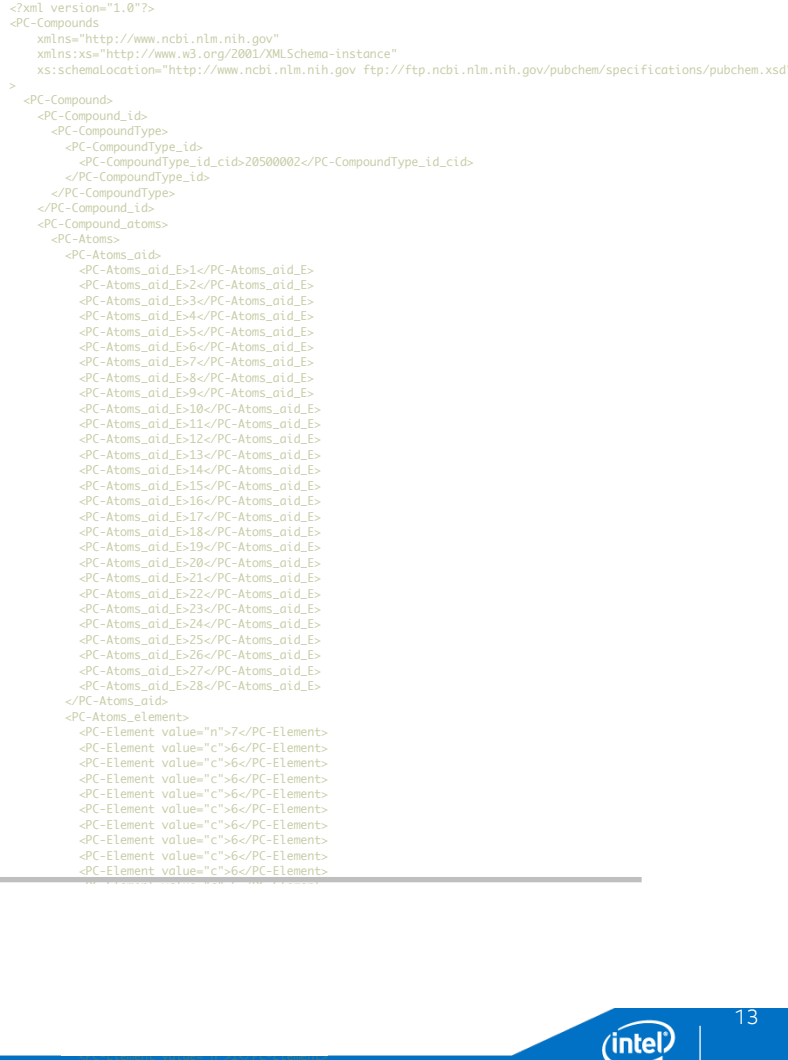
```
import trustedanalytics as ta
ta.CsvFile("raw_data.csv", schema=[("col1", ta.int32), ("col2", ta.float32)])
```

Be sure you have the correct delimiter!

XML Files

Parallel Ingest:

- Just point the toolkit to a directory containing multiple *xml* files to process them all at once.



XML Files

Parallel Ingest:

- Just point the toolkit to a directory containing multiple *xml* files to process them all at once.

Identify the tag denoting separation of entities:

```
<?xml version="1.0"?>
<PC-Compounds
    xmlns="http://www.ncbi.nlm.nih.gov"
    xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
    xs:schemaLocation="http://www.ncbi.nlm.nih.gov ftp://ftp.ncbi.nlm.nih.gov/pubchem/specifications/pubchem.xsd"
>
  <PC-Compound>
    <PC-Compound_id>
      <PC-CompoundType>
        <PC-CompoundType_id>
          <PC-CompoundType_id_cid>20500002</PC-CompoundType_id_cid>
        </PC-CompoundType_id>
      </PC-CompoundType>
    </PC-Compound_id>
    <PC-Compound_atoms>
      <PC-Atoms>
        <PC-Atoms_aid>
          <PC-Atoms_aid_E>1</PC-Atoms_aid_E>
          <PC-Atoms_aid_E>2</PC-Atoms_aid_E>
          <PC-Atoms_aid_E>3</PC-Atoms_aid_E>
          <PC-Atoms_aid_E>4</PC-Atoms_aid_E>
          <PC-Atoms_aid_E>5</PC-Atoms_aid_E>
          <PC-Atoms_aid_E>6</PC-Atoms_aid_E>
          <PC-Atoms_aid_E>7</PC-Atoms_aid_E>
          <PC-Atoms_aid_E>8</PC-Atoms_aid_E>
          <PC-Atoms_aid_E>9</PC-Atoms_aid_E>
          <PC-Atoms_aid_E>10</PC-Atoms_aid_E>
          <PC-Atoms_aid_E>11</PC-Atoms_aid_E>
          <PC-Atoms_aid_E>12</PC-Atoms_aid_E>
          <PC-Atoms_aid_E>13</PC-Atoms_aid_E>
          <PC-Atoms_aid_E>14</PC-Atoms_aid_E>
          <PC-Atoms_aid_E>15</PC-Atoms_aid_E>
          <PC-Atoms_aid_E>16</PC-Atoms_aid_E>
          <PC-Atoms_aid_E>17</PC-Atoms_aid_E>
          <PC-Atoms_aid_E>18</PC-Atoms_aid_E>
          <PC-Atoms_aid_E>19</PC-Atoms_aid_E>
          <PC-Atoms_aid_E>20</PC-Atoms_aid_E>
          <PC-Atoms_aid_E>21</PC-Atoms_aid_E>
          <PC-Atoms_aid_E>22</PC-Atoms_aid_E>
          <PC-Atoms_aid_E>23</PC-Atoms_aid_E>
          <PC-Atoms_aid_E>24</PC-Atoms_aid_E>
          <PC-Atoms_aid_E>25</PC-Atoms_aid_E>
          <PC-Atoms_aid_E>26</PC-Atoms_aid_E>
          <PC-Atoms_aid_E>27</PC-Atoms_aid_E>
          <PC-Atoms_aid_E>28</PC-Atoms_aid_E>
        </PC-Atoms_aid>
        <PC-Atoms_element>
          <PC-Element value="n">7</PC-Element>
          <PC-Element value="c">6</PC-Element>
          <PC-Element value="c">6</PC-Element>
          <PC-Element value="c">6</PC-Element>
          <PC-Element value="c">6</PC-Element>
          <PC-Element value="c">6</PC-Element>
          <PC-Element value="c">6</PC-Element>
          <PC-Element value="c">6</PC-Element>
          <PC-Element value="c">6</PC-Element>
          <PC-Element value="c">6</PC-Element>
          <PC-Element value="c">6</PC-Element>
```

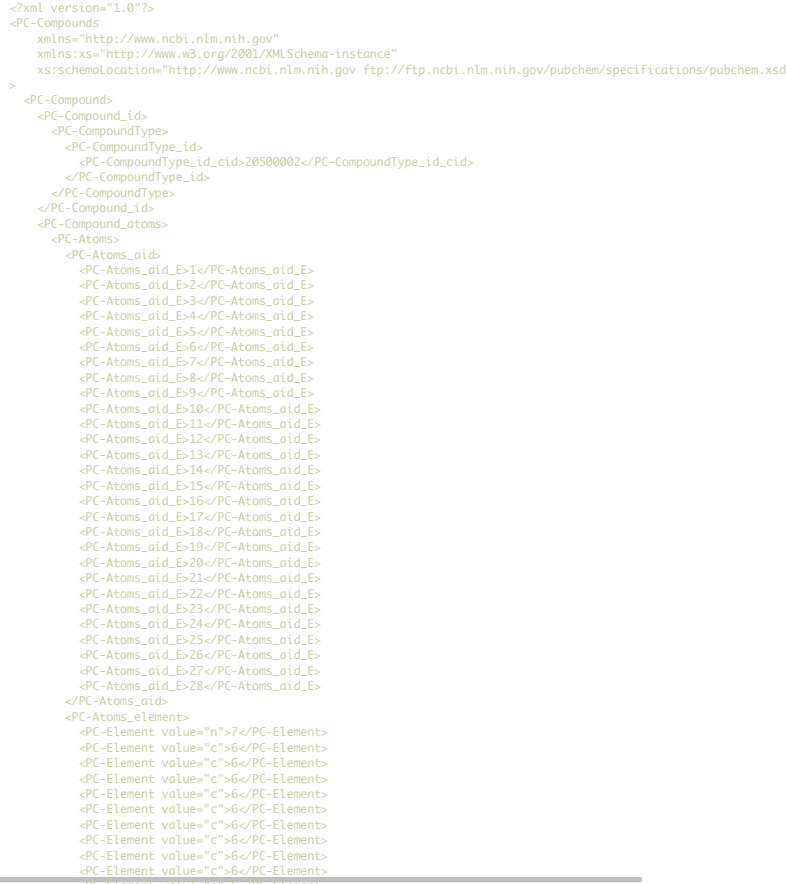
XML Files

Parallel Ingest:

- Just point the toolkit to a directory containing multiple *xml* files to process them all at once.

Identify the tag denoting separation of entities:

```
<note>
   <to>Self</to>
   <from>Self</from>
   <heading>Eagles Dig</heading>
   <body>Chip Kelly was better at UO</body>
</note>
```



XML Files

Parallel Ingest:

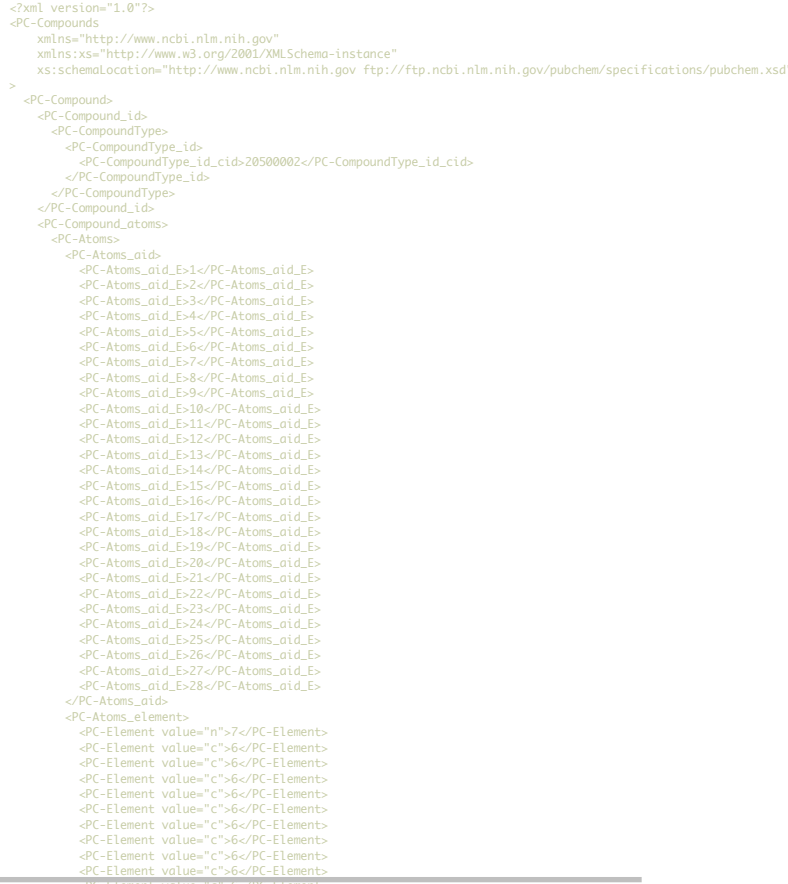
- Just point the toolkit to a directory containing multiple *xml* files to process them all at once.

Identify the tag denoting separation of entities:

```
<note>
   <to>Self</to>
   <from>Self</from>
   <heading>Eagles Dig</heading>
   <body>Chip Kelly was better at UO</body>
</note>
```

Example:

```
xml = ia.XmlFile(file_name='path', tag_name='tag')
frame = ia.Frame(source=xml, name='name')
```



XML Files

Parallel Ingest:

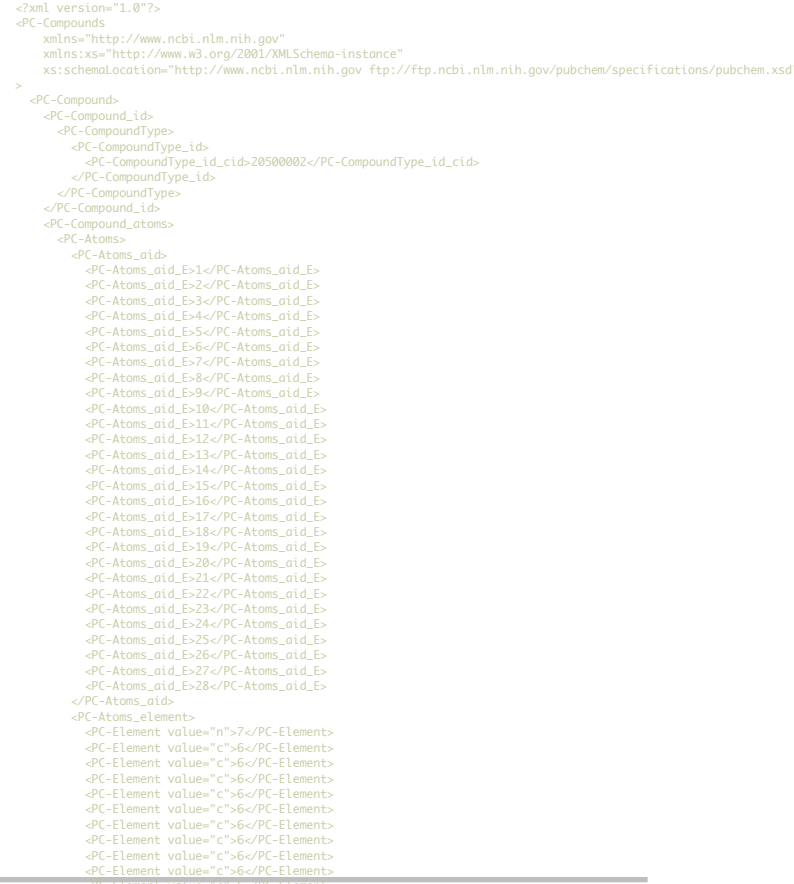
- Just point the toolkit to a directory containing multiple *xml* files to process them all at once.

Identify the tag denoting separation of entities:

```
<note>
   <to>Self</to>
   <from>Self</from>
   <heading>Eagles Dig</heading>
   <body>Chip Kelly was better at UO</body>
</note>
```

Example:

```
xml = ia.XmlFile(file_name='path', tag_name='tag')
frame = ia.Frame(source=xml, name='name')
```



Schema-naïve processing is currently a WIP. It will make everyone's life easier!

XML Fil

¡Pro Tip!

Pa gest:

Just point
multiple *x*

Identify the tag

```
<note>
   <to>Self</
   <from>Self
   <heading>E
   <body>Chip
</note>
```

Wrapping this sort of processing code in a function will be useful when working with complex xml files!

```
def parse_xml_to_frame(path, tag, name):
    """
    Helper function to convert an xml file on the hdfs into a data frame...
    """
    xml = ia.XmlFile(path, tag)

    # Check that the frame doesn't already exist. Drop it, if it does...
    if name in ia.get_frame_names():
        sys.stderr.write("Dropping existing frame named {NAME}...\n".format(NAME=name))
        ia.drop_frames(name)
    frame = ia.Frame(xml, name=name)
    return frame
```

Example:

```
xml = ia.XmlFile(file_name='path', tag_name='tag')
frame = ia.Frame(source=xml, name='name')
```

Schema-naïve processing is currently a WIP. It will make everyone's life easier!

JSON Files

Parallel Ingest

- Just point the toolkit to a directory containing multiple *json* files to process them all at once.
- Creating a frame from *json* will result in a frame with a single column: '`data_lines`'

ia.JsonFile:

```
__init__(self, file_name)
```

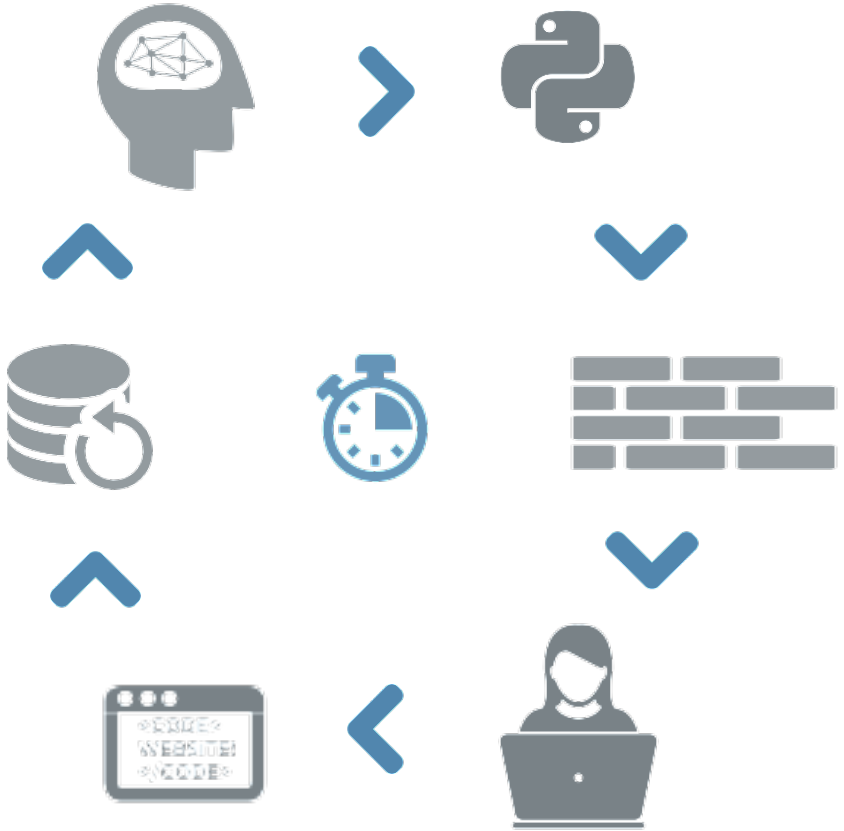
Example:

```
json_file = ta.JsonFile("data/raw_data.json")
my_frame = ta.Frame(json_file)
```


```
{u'ADMITTING_CCS_LVL_1_LABELs': [u'Diseases of the respiratory system'],
 u'ADMITTING_CCS_LVL_2_LABELs': [u'Other lower respiratory disease [133.]'],
 u'ADMITTING_CCS_LVL_3_LABELs': [u'Other and unspecified lower respiratory disease'],
 u'ADMITTING_CCS_LVL_4_LABELs': [u' '],
 u'ADMITTING_DIAG_CCS_LVL_1_LABELs': [u'nan',
                                      u'Diseases of the respiratory system'],
 u'ADMITTING_DIAG_CCS_LVL_2_LABELs': [u'nan',
                                      u'Other lower respiratory disease [133.]'],
 u'ADMITTING_DIAG_CCS_LVL_3_LABELs': [u'nan',
                                      u'Other and unspecified lower respiratory disease'],
 u'ADMITTING_DIAG_CCS_LVL_4_LABELs': [u'nan', u' '],
 u'ADMITTING_DIAG_CODEs': [u'786.05'],
 u'ADMITTING_DIAG_DESCs': [u'Shortness of breath'],
 u'ADMIT_REASONs': [u'COPD EXACERBATION', u'SICK0985'],
 u'ADMIT_SERVICE_CODE': u'GER',
 u'ADMIT_SERVICE_DESCRIPTION': u'GERIATRIC MEDICINE',
 u'ADMIT_SERVICE_GROUP': u'MEDICINE',
 u'ADMIT_SOURCE': u'Emergency Room',
 u'ADMIT_SOURCE_MEDVIEW': u'EMERGENCY OP UNIT',
 u'ADMIT_TYPE': u'Emergency',
 u'ADM_DATE': {u'$date': 1557447300000},
 u'AGE': 77,
 u'ARITHMETIC_LOS': 4.2,
 u'CCS_LVL_1_LABELs': [u'Diseases of the respiratory system',
                       u'Diseases of the circulatory system',
                       u'Endocrine; nutritional; and metabolic diseases and immunity disorders'
                       u'Diseases of the blood and blood-forming organs'],
 u'CCS_LVL_2_LABELs': [u'Chronic obstructive pulmonary disease and bronchiectasis [127.]',
                       u'Diseases of the heart',
                       u'Diabetes mellitus without complication [49.]',
                       u'Gout and other crystal arthropathies [54.]',
                       u'Anemia',
                       u'Other lower respiratory disease [133.]'],
 u'CCS_LVL_3_LABELs': [u'Obstructive chronic bronchitis',
                       u'Congestive heart failure; nonhypertensive [108.]',
                       u' ',
                       u'Deficiency and other anemia [59.]',
                       u'Other and unspecified lower respiratory disease'],
```


For large, nested json, `data_lines` can get quite large, causing some operations to take time to complete

- 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

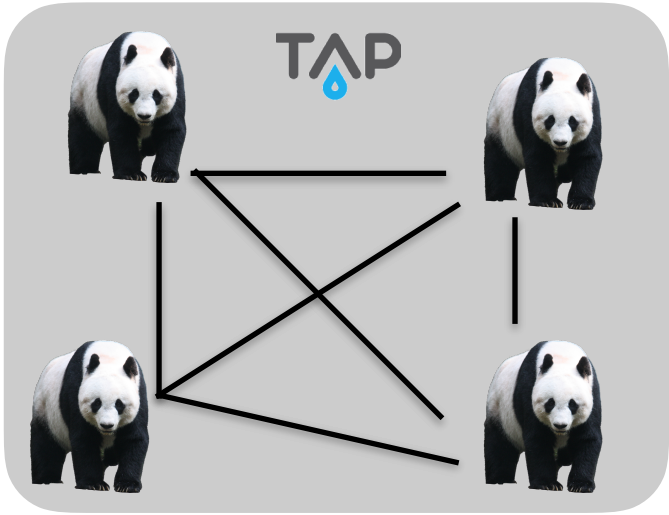
TAP Data Frames

- 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

— 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)
```

— 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(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

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)
```

```
>>> 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)

```
>> 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

```
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!

- Building a topic model
- Training a random forest (RF) model
- Classifying records with RF

questions?

code!

Problems:

[**1**] Create a 3*x1* frame with the following contents:

(c1) “animals”: three species of animal

[**2**] Ingest a new character-separated (csv) file (offsides.tsv)

[**3**] Create a 100 x 2 frame with the following contents:

(c1) “rn”: 100 random integers between 0-1000

(c2) “evenodd”: “EVEN” if the corresponding integer is even, otherwise “ODD”

Problems:

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

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

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

Problems:

[**7**] In the `inpat` data set there are multiple fields of the pattern 'FINAL_DIAG_CCS_LVL_n_LABELs', where *n* is a number. Use add_columns to extract these data as a single column.

[**8**] In the column updated in [1], there are sometimes codes mixed in with plain-english labels (e.g., "*Deficiency and other anemia [59.]*"). Add a second column that has removed these codes.

[**9**] Many unigrams in the labels are common, and many are potentially uninformative. Using a mixture of EDA and domain expertise, write your own stop word removal function to clean out uninformative unigrams.

Problem:

[**10**] Use a feature besides admit/discharge date to create a gold-standard label column (i.e., generate “POSITIVE” and “NEGATIVE” labels according to some reproducible rule).

[**11**] Build a Random Forest classifier using any combination of features you choose to classify on the gold standard label chosen in [1]. Evaluate.

[**12**] What steps would be involved in creating a function to perform cross-validation? Can you think of a way to implement this in TAP?