Pandas Under The Hood

Peeking behind the scenes of a high performance data analysis library

July 25, 2015 | Jeff Tratner (@jtratner)
Pandas - large, well-established project.

Flexible and powerful data analysis / manipulation library for Python, providing labeled data structures similar to R data.frame objects, statistical functions, and much more http://pandas.pydata.org

- **12,165 commits**
- **7 branches**
- **63 releases**
- **406 contributors**

**Aug 2, 2009 – Jul 23, 2015**

Contributions to master, excluding merge commits
Overview

Intro

Data in Python Background

Indexing

Getting and Storing Data

Fast Grouping / Factorizing

Summary
Overview

Intro

Data in Python Background

Indexing

Getting and Storing Data

Fast Grouping / Factorizing

Summary
Pandas - huge code base

- 200K lines of code
- Depends on many other libraries
- Goal: orient towards key internal concepts
Pandas community rocks!

- Created by Wes McKinney, now maintained by Jeff Reback and many others
- Really open to small contributors
- Many friendly and supportive maintainers
- Go contribute!
Pandas provides a flexible API for data

- **Dataframe** - 2D container for *labeled* data
- **Read data** (read_csv, read_excel, read_hdf, read_sql, etc)
- **Write data** (df.to_csv(), df.to_excel())
- **Select, filter, transform data**
- **Big emphasis on labeled data**
- **Works really nicely with other python data analysis libraries**

<table>
<thead>
<tr>
<th>Product</th>
<th>Quantity</th>
<th>Revenue</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>523</td>
<td>1103.25</td>
<td>5230</td>
</tr>
<tr>
<td>B</td>
<td>200</td>
<td>1525.10</td>
<td>860</td>
</tr>
<tr>
<td>C</td>
<td>148</td>
<td>3892.50</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>1610</td>
<td>5730.25</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>122</td>
<td>580.12</td>
<td>600</td>
</tr>
<tr>
<td>F</td>
<td>10</td>
<td>55342.00</td>
<td>100</td>
</tr>
</tbody>
</table>
Overview

Intro

Data in Python Background

Indexing

Getting and Storing Data

Fast Grouping / Factorizing

Summary
Python flexibility can mean slowness
Take a simple-looking operation...

```python
for i, x in enumerate(some_array):
    some_array[i] = log(x) * 5 + log(i)
```
Python’s dynamicity can be a problem

Have to lookup (i) and (log) repeatedly, even though they haven’t changed.
Python C-API lets you avoid overhead.

- Choose when you want to bubble up to Python level
- Get compiler optimizations like other C programs
- Way more control over memory management.
Bookkeeping on Python objects.

- PyObject_HEAD:
  - Reference Count
  - Type
  - Value (or pointer to value)

Illustration: Jake VanderPlas: Why Python is Slow
Poor memory locality in Python containers.

How can we make this better?

Illustration: [Jake VanderPlas: Why Python is Slow](#)
Pack everything together in a “C”-level array
Numpy enables efficient, vectorized operations on (nd)arrays.

- **ndarray** is a pointer to memory in C or Fortran
- Based on really sturdy code mostly written in Fortran
- Can stay at C-level if you vectorize operations and use specialized functions ('ufuncs')

Illustration: [Jake VanderPlas: Why Python is Slow](https://jakevdp.github.io/plotly/tutorials/python-performance/)
Cython lets you compile Python to C

- Compiles typed Python to C (preserving traceback!)
- Specialized for numpy
- Lots of goodies
  - Inline functions
  - Call c functions
  - Bubbles up to Python only when necessary

```python
def lookup(self, ndarray[int32_t] values):
    cdef:
        Py_ssize_t i, n = len(values)
        int ret = 0
        int32_t val
        khter_t k
        ndarray[int32_t] locs = np.empty(n, dtype=np.int64)

    for i in range(n):
        val = values[i]
        k = kh_get_int32(self.table, val)
        if k != self.table.n_buckets:
            locs[i] = self.table.vals[k]
        else:
            locs[i] = -1

    return locs
```
Example compiled Cython code

```c
/* "pandas/hashtable.pyx":328$*/
*-
* for i in range(n):$
  *   val = values[i] # <<<<<<<<<<<<<<$
  *   k = kh_get_int32(self.table, val)$
  * if k != self.table.n_buckets:$$
*/$
__pyx_t_9 = __pyx_v_i;$$
__pyx_t_10 = -1;$
if (__pyx_t_9 < 0) {$
  __pyx_t_9 += __pyx_pybuffernd_values.diminfo[0].shape;$
  if (unlikely(__pyx_t_9 < 0)) __pyx_t_10 = 0;$
} else if (unlikely(__pyx_t_9 >= __pyx_pybuffernd_values.diminfo[0].shape)) __pyx_t_10 = 0;$
if (unlikely(__pyx_t_10 != -1)) {$
  _Pyx_RaiseBufferIndexError(__pyx_t_10);$$_pyx_filename = __pyx_f[0]; __pyx_lineno = 328; __pyx_clineno = __LINE__; goto __pyx_l1_error;}$$
__pyx_v_val = (___Pyx_BufPtrStrided1d(__pyx_t_5numpy_int32_t *, __pyx_pybuffernd_values.rcbuffer->pybuffer.buf, __pyx_t_9, __pyx_t_10, __pyx_t_10) & (((1 << 31) - 1) << 31)) + __pyx_t_9;$$
/* "pandas/hashtable.pyx":329$*/$
* k = kh_get_int32(self.table, val) # <<<<<<<<<<<<<<$
*/$
__pyx_v_k = kh_get_int32(__pyx_v_self->table, __pyx_v_val);$$
/* "pandas/hashtable.pyx":330$*/$
* if k != self.table.n_buckets: # <<<<<<<<<<<<<<$
*   locs[i] = self.table.vals[k]$$$
/* "pandas/hashtable.pyx":331$*/$
* if k != self.table.n_buckets:$
*   locs[i] = self.table.vals[k] # <<<<<<<<<<<<<<$
/* "$pandas/hashtable.pyx":332$*/$
__pyx_t_11 = ((__pyx_v_k != __pyx_v_self->table->n_buckets) != 0);$$
if (__pyx_t_11) {$$
/* "$pandas/hashtable.pyx":333$*/$
* if k != self.table.n_buckets:$
*   locs[i] = self.table.vals[k]$$$
/* "$pandas/hashtable.pyx":334$*/$
__pyx_t_12 = __pyx_v_i;$$
__pyx_t_10 = -1;$
if (__pyx_t_12 < 0) {$$
```
Numexpr - compiling Numpy bytecode for better performance.

- Compiles bytecode on numpy arrays to optimized ops
- Chunks numpy arrays and runs operations in cache-optimized groups
- Less overhead from temporary arrays

```python
for i in xrange(0, len(a), 256):
    r0 = a[i:i+128]
    r1 = b[i:i+128]
    multiply(r0, 2, r2)
    multiply(r1, 3, r3)
    add(r2, r3, r2)
    c[i:i+128] = r2
```
So...why pandas?
Pandas enables flexible, performant analysis.

- Heterogenous data types
- Easy, fast missing data handling
- Easier to write generic code
- *Labeled* data (numpy mostly assumes index == label)
- *Relational* data
Overview

Intro Pandas

Data in Python Background

Indexing

Getting and Storing Data

Fast Grouping / Factorizing

Summary
Core pandas data structure is the DataFrame

<table>
<thead>
<tr>
<th>Product</th>
<th>Quantity</th>
<th>Revenue</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>523</td>
<td>1103.25</td>
<td>5230</td>
</tr>
<tr>
<td>B</td>
<td>200</td>
<td>1525.10</td>
<td>860</td>
</tr>
<tr>
<td>C</td>
<td>148</td>
<td>3892.50</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>1610</td>
<td>5730.25</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>122</td>
<td>580.12</td>
<td>600</td>
</tr>
<tr>
<td>F</td>
<td>10</td>
<td>55342.00</td>
<td>100</td>
</tr>
</tbody>
</table>

- Indexes
- Blocks of Data
- Columns are “Series” (1 dimensional NDFrame)
Indexing Basics
Indexes are a big mapping

- Essentially a big dict
- (set of) label(s) → integer locations
- read as “row C” maps to location 2
- “metadata” on DataFrame
- Any Series of Data can be converted to an Index
- Immutable!
Index task 1: Lookups (map labels to locations)

<table>
<thead>
<tr>
<th>Product</th>
<th>Quantity</th>
<th>Revenue</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>523</td>
<td>1103.25</td>
<td>5230</td>
</tr>
<tr>
<td>B</td>
<td>200</td>
<td>1525.10</td>
<td>860</td>
</tr>
<tr>
<td>C</td>
<td>148</td>
<td>3892.50</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>1610</td>
<td>5730.25</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>122</td>
<td>580.12</td>
<td>600</td>
</tr>
<tr>
<td>F</td>
<td>10</td>
<td>55342.00</td>
<td>100</td>
</tr>
</tbody>
</table>

\[ \text{df1.loc['C']} \]

Quantity 148.0
Revenue 3892.5
Points 0.0
Name: C, dtype: float64
## Index task 2: Enable combining objects

<table>
<thead>
<tr>
<th>Product</th>
<th>Quantity</th>
<th>Revenue</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>523</td>
<td>1103.25</td>
<td>5230</td>
</tr>
<tr>
<td>B</td>
<td>200</td>
<td>1525.10</td>
<td>860</td>
</tr>
<tr>
<td>C</td>
<td>148</td>
<td>3892.50</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>1610</td>
<td>5730.25</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>122</td>
<td>580.12</td>
<td>600</td>
</tr>
<tr>
<td>F</td>
<td>10</td>
<td>55342.00</td>
<td>100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product</th>
<th>Quantity</th>
<th>Revenue</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>A</td>
<td>100</td>
<td>22.50</td>
</tr>
<tr>
<td>C</td>
<td>200</td>
<td>540.25</td>
</tr>
<tr>
<td>B</td>
<td>300</td>
<td>1534.00</td>
</tr>
<tr>
<td>E</td>
<td>400</td>
<td>2134.00</td>
</tr>
</tbody>
</table>

- Translate between different indexes and columns
- Numpy ops don’t know about labels
- Make objects compatible for numpy ops
Example: Arithmetic

<table>
<thead>
<tr>
<th>Product</th>
<th>Quantity</th>
<th>Revenue</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>523</td>
<td>1103.25</td>
<td>5230</td>
</tr>
<tr>
<td>B</td>
<td>200</td>
<td>1525.10</td>
<td>860</td>
</tr>
<tr>
<td>C</td>
<td>148</td>
<td>3892.50</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>1610</td>
<td>5730.25</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>122</td>
<td>580.12</td>
<td>600</td>
</tr>
<tr>
<td>F</td>
<td>10</td>
<td>55342.00</td>
<td>100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product</th>
<th>Quantity</th>
<th>Revenue</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>A</td>
<td>100</td>
<td>22.50</td>
</tr>
<tr>
<td>C</td>
<td>200</td>
<td>540.25</td>
</tr>
<tr>
<td>B</td>
<td>300</td>
<td>1534.00</td>
</tr>
<tr>
<td>E</td>
<td>400</td>
<td>2134.00</td>
</tr>
</tbody>
</table>

+ =

<table>
<thead>
<tr>
<th>Product</th>
<th>Quantity</th>
<th>Revenue</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>623</td>
<td>1125.75</td>
<td>NaN</td>
</tr>
<tr>
<td>B</td>
<td>500</td>
<td>3059.10</td>
<td>NaN</td>
</tr>
<tr>
<td>C</td>
<td>348</td>
<td>4432.75</td>
<td>NaN</td>
</tr>
<tr>
<td>D</td>
<td>1610</td>
<td>5730.25</td>
<td>NaN</td>
</tr>
<tr>
<td>E</td>
<td>522</td>
<td>2714.12</td>
<td>NaN</td>
</tr>
<tr>
<td>F</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
Align the index of second DataFrame (get_indexer)

<table>
<thead>
<tr>
<th>df1 index</th>
<th>df2 index</th>
<th>Aligned</th>
<th>Aligned version of df2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>D</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>B</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>×</td>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

(lookup value of first index on other index)

<table>
<thead>
<tr>
<th>Product</th>
<th>Quantity</th>
<th>Revenue</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100</td>
<td>22.50</td>
<td>NaN</td>
</tr>
<tr>
<td>B</td>
<td>300</td>
<td>1534.00</td>
<td>NaN</td>
</tr>
<tr>
<td>C</td>
<td>200</td>
<td>540.25</td>
<td>NaN</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0.00</td>
<td>NaN</td>
</tr>
<tr>
<td>E</td>
<td>400</td>
<td>2134.00</td>
<td>NaN</td>
</tr>
<tr>
<td>F</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
Scaling up...
Indexes have to do tons of lookups - needs to be fast!

- Answer: Klib!
- Super fast dict implementation specialized for each type (int, float, object, etc)
- Pull out an entire ndarray worth of values basically without bubbling up to Python level
- e.g., kh_get_int32, kh_get_int64, etc.
Overview

Intro Pandas
Data in Python Background
Indexing
Getting and Storing Data
Fast Grouping / Factorizing
Summary
Converting data

<table>
<thead>
<tr>
<th>Timestamp, UID, Product, Quantity, Member, Promo, Points, Paid</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015-07-01 02:50:00, xgy7b, A, 3, True, SEA15, 30, 21.5</td>
</tr>
<tr>
<td>2015-07-01 03:30:00, sot5y, C, 1, False, NA, 10, 15.0</td>
</tr>
<tr>
<td>2015-07-02 03:52:00, g8z8l, B, 2, False, NA, 30, 10.25</td>
</tr>
<tr>
<td>2015-07-03 04:01:00, lxzuo, A, 2, False, SEA15, 15, 16.25</td>
</tr>
<tr>
<td>2015-07-03 05:30:00, 3peyj, C, 4, True, BOB10, 10, 28.5</td>
</tr>
</tbody>
</table>
Getting in data: convert to Python, coerce types.

- **CSV - C and Python engine**
  - C engine: specialized reader that can read a subset of columns and handle comments / headers in low memory (fewer intermediate python objects)
  - iterate over possible dtypes and try converting to each one on all rows / subset of rows (dates, floats, integers, NA values, etc)

- **Excel**
  - use an external library, take advantage of hinting
  - uses TextParser Python internals
Storing Data - Blocks
Data is split into blocks under the hood

DataFrame

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>UID</th>
<th>Product</th>
<th>Quantity</th>
<th>Member</th>
<th>Promo</th>
<th>Points</th>
<th>Paid</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015-07-01 02:50:00</td>
<td>xgy7b</td>
<td>A</td>
<td>3</td>
<td>True</td>
<td>SEA15</td>
<td>30</td>
<td>21.50</td>
</tr>
<tr>
<td>2015-07-01 03:30:00</td>
<td>sot5y</td>
<td>C</td>
<td>1</td>
<td>False</td>
<td>NaN</td>
<td>10</td>
<td>15.00</td>
</tr>
<tr>
<td>2015-07-02 03:52:00</td>
<td>g8z8l</td>
<td>B</td>
<td>2</td>
<td>False</td>
<td>NaN</td>
<td>30</td>
<td>10.25</td>
</tr>
<tr>
<td>2015-07-03 04:01:00</td>
<td>lxzuo</td>
<td>A</td>
<td>2</td>
<td>False</td>
<td>SEA15</td>
<td>15</td>
<td>16.25</td>
</tr>
<tr>
<td>2015-07-03 05:30:00</td>
<td>3peyj</td>
<td>C</td>
<td>4</td>
<td>True</td>
<td>BOB10</td>
<td>10</td>
<td>28.50</td>
</tr>
</tbody>
</table>
BlockManager handles translation between DataFrame and blocks

- **BlockManager**
  - Manages axes (indexes)
  - Getting and changing data
  - DataFrame -> high level API

- **Blocks**
  - Specialized by type
  - Only cares about locations
  - Usually operating within types with NumPy
Implications: within dtypes ops are fine

- Slicing within a dtype no copy
  - `df.loc[:'2015-07-03', ['quantity', 'points']]`

- cross-dtype slicing generally requires copy

- SettingWithCopy
  - not sure if you’re referencing same underlying info
Implications: fixed size blocks make appends expensive

- Have to copy and resize all blocks on append*
- Various strategies to deal with this
  - zero out space to start
  - pull everything into Python first
  - concatenate multiple frames

* This means **multiple** appends (concat & append are equivalent here). I.e., better to join two big DataFrames than append each row individually.
Overview

Intro Pandas

Data in Python Background

Indexing

Getting and Storing Data

Fast Factorizing / Grouping

Summary
Factorizing underlies key pandas ops

- Mapping of repeated keys → integer
- More efficient for memory & algorithms
- Used in a bunch of places
  - GroupBy
  - Hierarchical Indexes
  - Categoricals
- Klib again for fast dicts and lookups

```python
indexes = []
factors = {}
max_factor = 0
for elem in array:
    if elem not in factors:
        factors[elem] = max_factor
        max_factor += 1
    indexes.append(factors[elem])
```
Motivation: Counting Sort (or “group sort”)

- Imagine you have 100k rows, but only 10k unique values
- Instead of comparisons (O(NlogN)), can scan through, grab unique values and the count of how many times each value occurs
- now you know bin size and bin order
Handling more complicated situations

- E.g., multiple columns
- Factorize each one independently
- Compute cross product (can be really big!)
- Factorize again to compute space
With factors, more things are easy

- Only compute factors once (expensive!)
- Quickly subset in O(N) scans
- Easier to write type-specialized aggregation functions in Cython
Overview

Intro Pandas

Data in Python Background

Indexing

Getting and Storing Data

Fast Grouping / Factorizing

Summary
Summary

- The key to doing many small operations in Python: don’t do them in Python!
- Indexing: set-like ops, build mapping behind the scenes, powers high level API
- Blocks: Subsetting/changing/getting data
  - underlying structure helps you think about when copies are going to happen
  - but copies happen a lot
- (Fast) factorization underlies many important operations
Thanks!

@jtratner on Twitter/Github
jeffrey.tratner@gmail.com