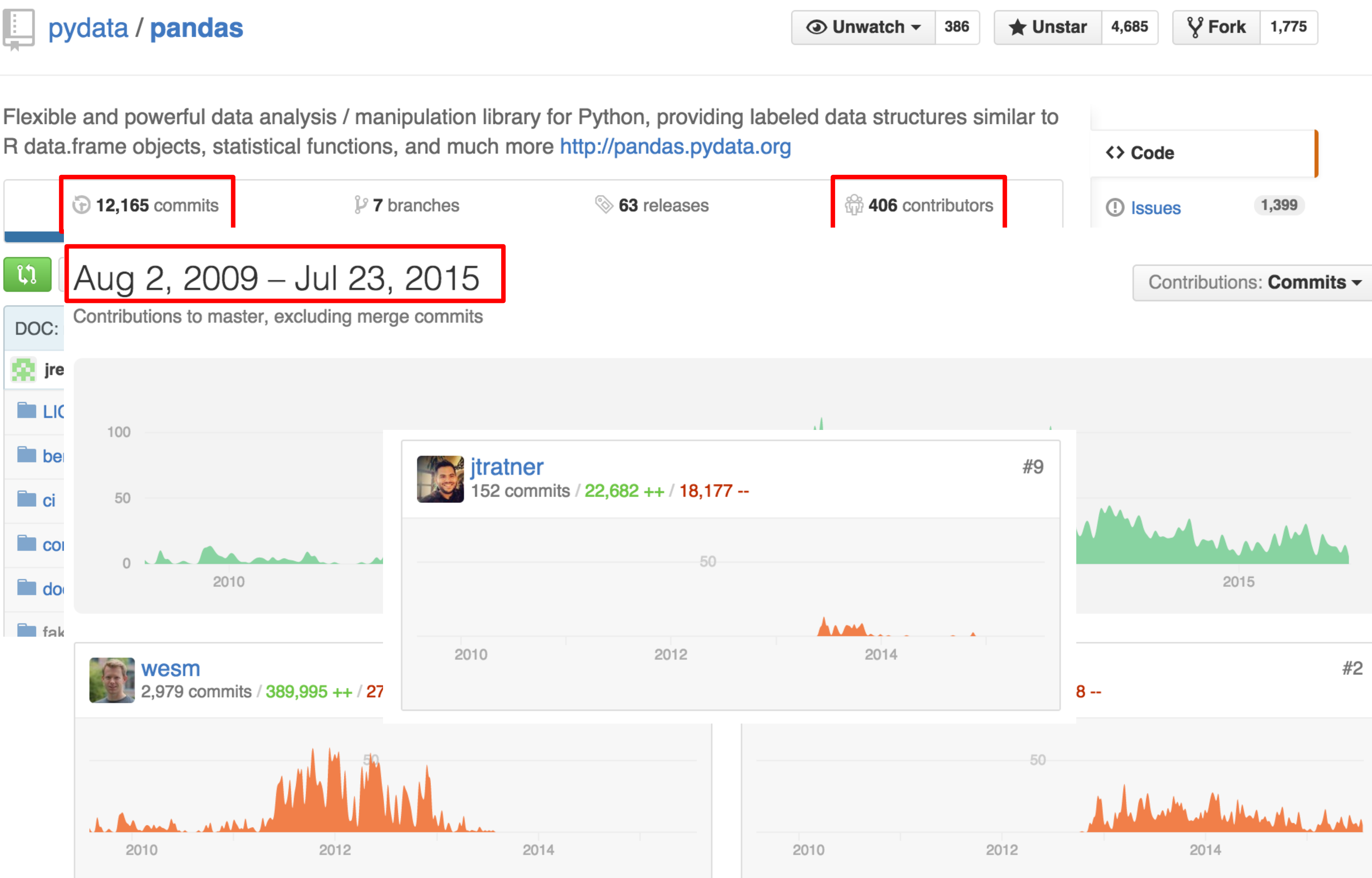
Pandas Under The Hood

Peeking behind the scenes of a high performance data analysis library

July 25, 2015 | Jeff Tratner (@jtratner)

Counsyl

Pandas - large, well-established project.



Overview

Intro

Data in Python Background

Indexing

Getting and Storing Data

Fast Grouping / Factorizing

Summary

Overview

Intro

Data in Python Background

Indexing

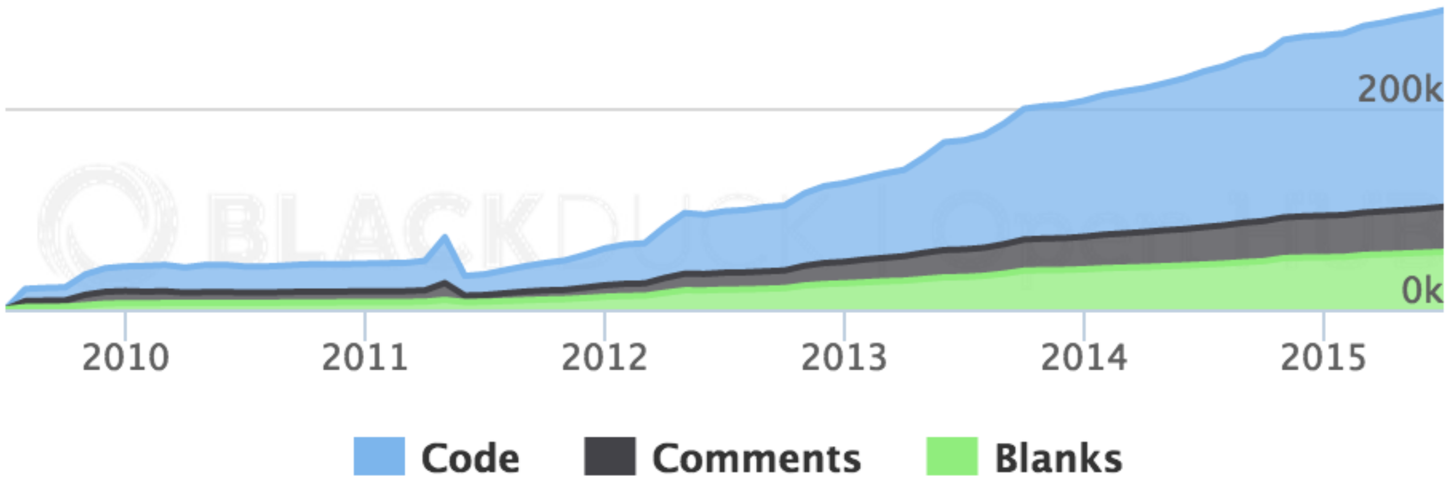
Getting and Storing Data

Fast Grouping / Factorizing

Summary

Pandas - huge code base

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



[Open Hub - Py-Pandas]()

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

	Quantity	Revenue	Points
Product			
A	523	1103.25	5230
B	200	1525.10	860
C	148	3892.50	0
D	1610	5730.25	0
E	122	580.12	600
F	10	55342.00	100

- 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

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...

```
for i, x in enumerate(some_array):
    some_array[i] = log(x) * 5 + log(i)
```

Python's dynamicity can be a problem

```
for i, x in enumerate(some_array):
    some_array[i] = log(x) * 5 + log(i)
```

dis.dis(<code>)

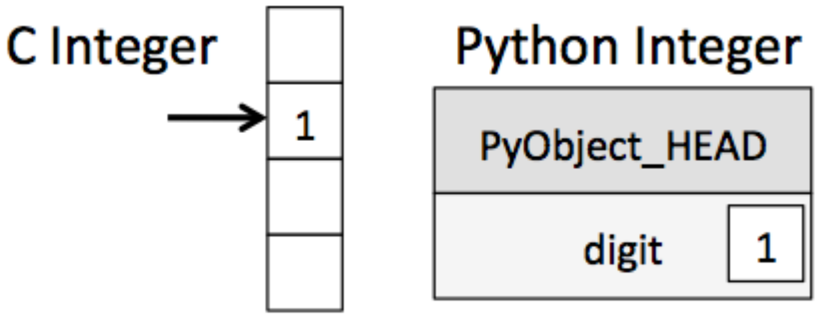
```
2            0 SETUP_LOOP              56 (to 59)
             3 LOAD_NAME                0 (enumerate)
             6 LOAD_NAME                1 (some_array)
             9 CALL_FUNCTION            1
            12 GET_ITER
      >>    13 FOR_ITER                42 (to 58)
            16 UNPACK_SEQUENCE          2
            19 STORE_NAME               2 (i)
            22 STORE_NAME               3 (x)

3           25 LOAD_NAME                4 (log)
            28 LOAD_NAME                3 (x)
            31 CALL_FUNCTION            1
            34 LOAD_CONST               0 (5)
            37 BINARY_MULTIPLY
            38 LOAD_NAME                4 (log)
            41 LOAD_NAME                2 (i)
            44 CALL_FUNCTION            1
            47 BINARY_ADD
            48 LOAD_NAME                1 (some_array)
            51 LOAD_NAME                2 (i)
            54 STORE_SUBSCR
            55 JUMP_ABSOLUTE           13
      >>    58 POP_BLOCK
      >>    59 LOAD_CONST               1 (None)
            62 RETURN_VALUE
```

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.

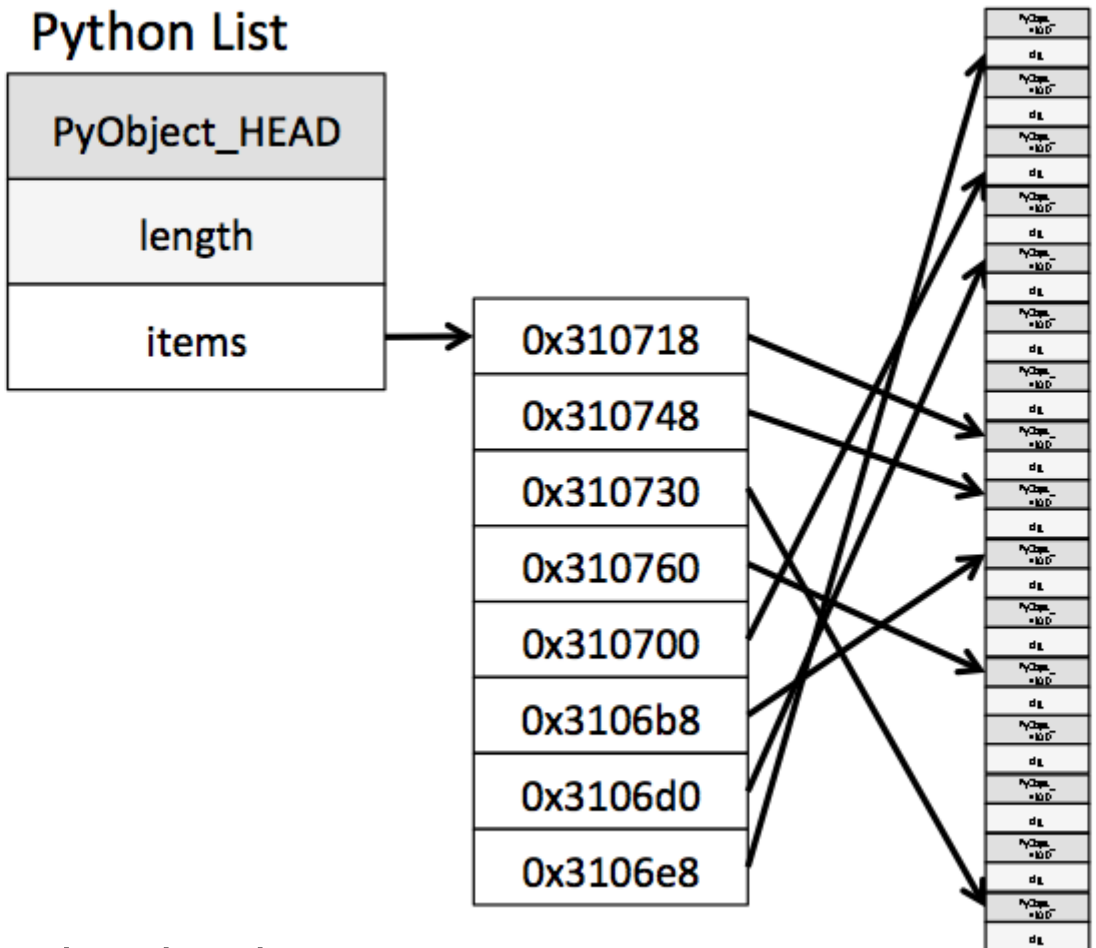
Bookkeeping on Python objects.



- 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

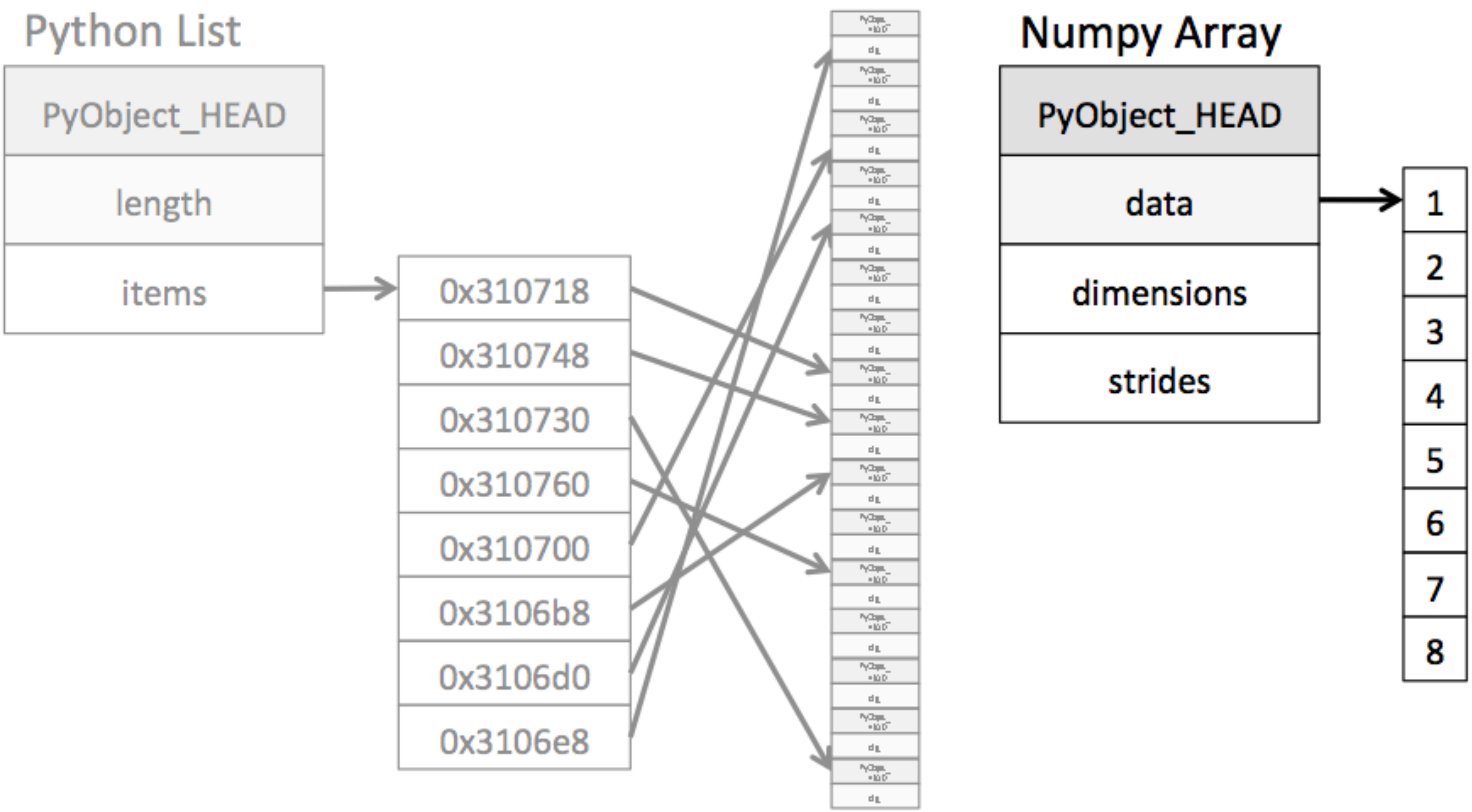
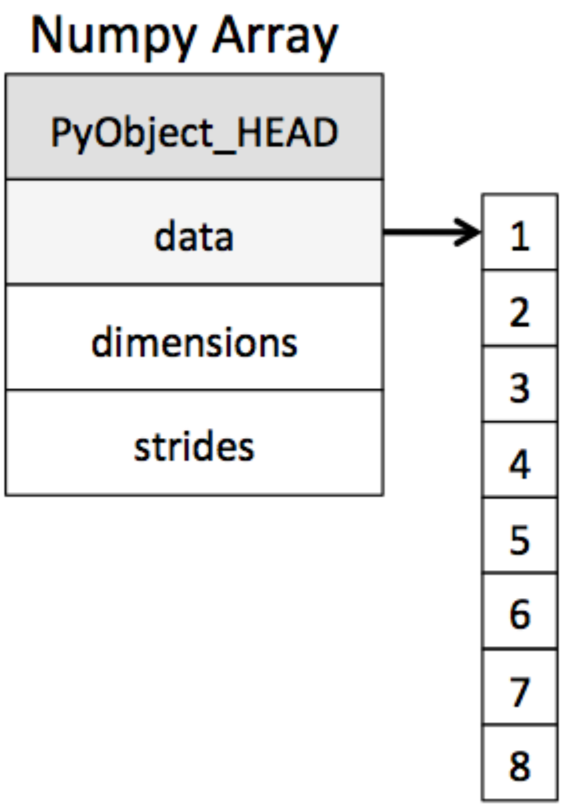


Illustration: [*Jake VanderPlas: Why Python is Slow*]()

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]()

Cython lets you compile Python to C

```
def lookup(self, ndarray[int32_t] values):$
    cdef:$
        Py_ssize_t i, n = len(values)$
        int ret = 0$
        int32_t val$
        khiter_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$
```

- 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

Example compiled Cython code

```
  /* "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_
$
  /* "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]$
 */$
  __pyx_t_11 = ((__pyx_v_k != __pyx_v_self->table->n_buckets) != 0);$
  if (__pyx_t_11) {$
$
    /* "pandas/hashtable.pyx":331$
 *             if k != self.table.n_buckets:$
 *                 locs[i] = self.table.vals[k]             # <<<<<<<<<<<<<<$
 */$
    __pyx_t_12 = __pyx_v_i;$
    __pyx_t_10 = -1;$
    if (__pyx_t_12 < 0) {$
```

Numexpr - compiling Numpy bytecode for better performance.

```
2*a + 3*b
```

```
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
```

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

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

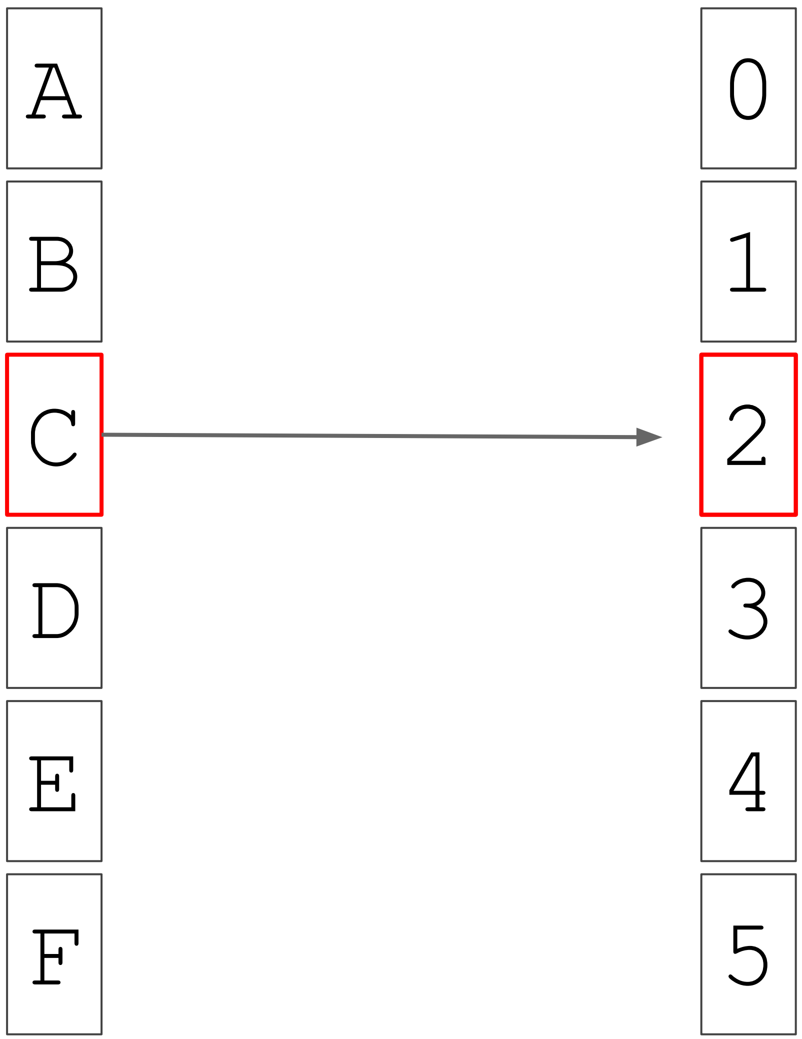
Core pandas data structure is the DataFrame

	Quantity	Revenue	Points
Product			
A	523	1103.25	5230
B	200	1525.10	860
C	148	3892.50	0
D	1610	5730.25	0
E	122	580.12	600
F	10	55342.00	100

- 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)

DF1

	Quantity	Revenue	Points
Product			
A	523	1103.25	5230
B	200	1525.10	860
C	148	3892.50	0
D	1610	5730.25	0
E	122	580.12	600
F	10	55342.00	100

```
df1.loc['C']
```

```
Quantity       148.0
Revenue       3892.5
Points           0.0
Name: C, dtype: float64
```

Index task 2: Enable combining objects

	Quantity	Revenue	Points
Product			
A	523	1103.25	5230
B	200	1525.10	860
C	148	3892.50	0
D	1610	5730.25	0
E	122	580.12	600
F	10	55342.00	100

	Quantity	Revenue
Product		
D	0	0.00
A	100	22.50
C	200	540.25
B	300	1534.00
E	400	2134.00

- Translate between different indexes and columns
- Numpy ops don't know about labels
- Make objects compatible for numpy ops

Example: Arithmetic

	Quantity	Revenue	Points
Product			
A	523	1103.25	5230
B	200	1525.10	860
C	148	3892.50	0
D	1610	5730.25	0
E	122	580.12	600
F	10	55342.00	100

+

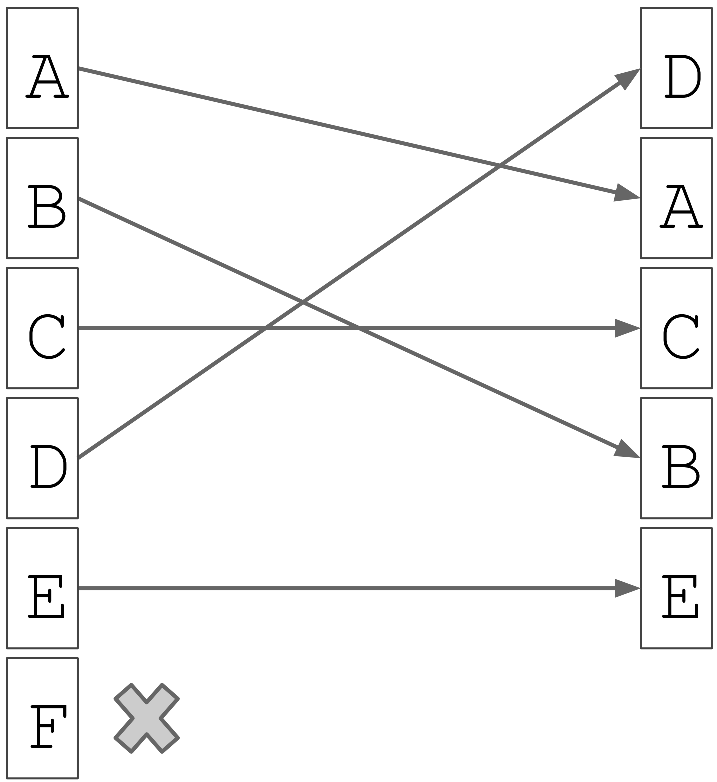
	Quantity	Revenue
Product		
D	0	0.00
A	100	22.50
C	200	540.25
B	300	1534.00
E	400	2134.00

=

	Quantity	Revenue	Points
Product			
A	623	1125.75	NaN
B	500	3059.10	NaN
C	348	4432.75	NaN
D	1610	5730.25	NaN
E	522	2714.12	NaN
F	NaN	NaN	NaN

Align the index of second DataFrame (get_indexer)

df1 index | df2 index | Aligned | Aligned version of df2



Aligned:

1
3
2
0
4
-1

	Quantity	Revenue	Points
Product			
A	100	22.50	NaN
B	300	1534.00	NaN
C	200	540.25	NaN
D	0	0.00	NaN
E	400	2134.00	NaN
F	NaN	NaN	NaN

(lookup value of first index on other index)

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.

```
  /* "pandas/hashtable.pyx":329$
 *             k = kh_get_int32(self.table, val)            # <<<<<<<<<<<<<<$
 */$
  __pyx_v_k = kh_get_int32(__pyx_v_self->table, __pyx_v_val);$
$
```

Overview

Intro Pandas

Data in Python Background

Indexing

Getting and Storing Data

Fast Grouping / Factorizing

Summary

Converting data

```
Timestamp,UID,Product,Quantity,Member,Promo,Points,Paid
2015-07-01 02:50:00,xgy7b,A,3,True,SEA15,30,21.5
2015-07-01 03:30:00,sot5y,C,1,False,NA,10,15.0
2015-07-02 03:52:00,g8z8l,B,2,False,NA,30,10.25
2015-07-03 04:01:00,lxzuo,A,2,False,SEA15,15,16.25
2015-07-03 05:30:00,3peyj,C,4,True,BOB10,10,28.5
```

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

	UID	Product	Quantity	Member	Promo	Points	Paid
Timestamp							
2015-07-01 02:50:00	xgy7b	A	3	True	SEA15	30	21.50
2015-07-01 03:30:00	sot5y	C	1	False	NaN	10	15.00
2015-07-02 03:52:00	g8z8l	B	2	False	NaN	30	10.25
2015-07-03 04:01:00	lxzuo	A	2	False	SEA15	15	16.25
2015-07-03 05:30:00	3peyj	C	4	True	BOB10	10	28.50

	0
0	21.50
1	15.00
2	10.25
3	16.25
4	28.50

	0	1	2
0	xgy7b	A	SEA15
1	sot5y	C	NaN
2	g8z8l	B	NaN
3	lxzuo	A	SEA15
4	3peyj	C	BOB10

	0	1
0	3	30
1	1	10
2	2	30
3	2	15
4	4	10

	0
0	True
1	False
2	False
3	False
4	True

BlockManager handles translation between DataFrame and blocks

BlockManager

Axes

Timestamp
2015-07-01 02:50:00
2015-07-01 03:30:00
2015-07-02 03:52:00
2015-07-03 04:01:00
2015-07-03 05:30:00

UID
Product
Quantity
Member
Promo
Points
Paid

Blocks

	0	1	2
0	xgy7b	A	SEA15
1	sot5y	C	NaN
2	g8z8l	B	NaN
3	lxzuo	A	SEA15
4	3peyj	C	BOB10

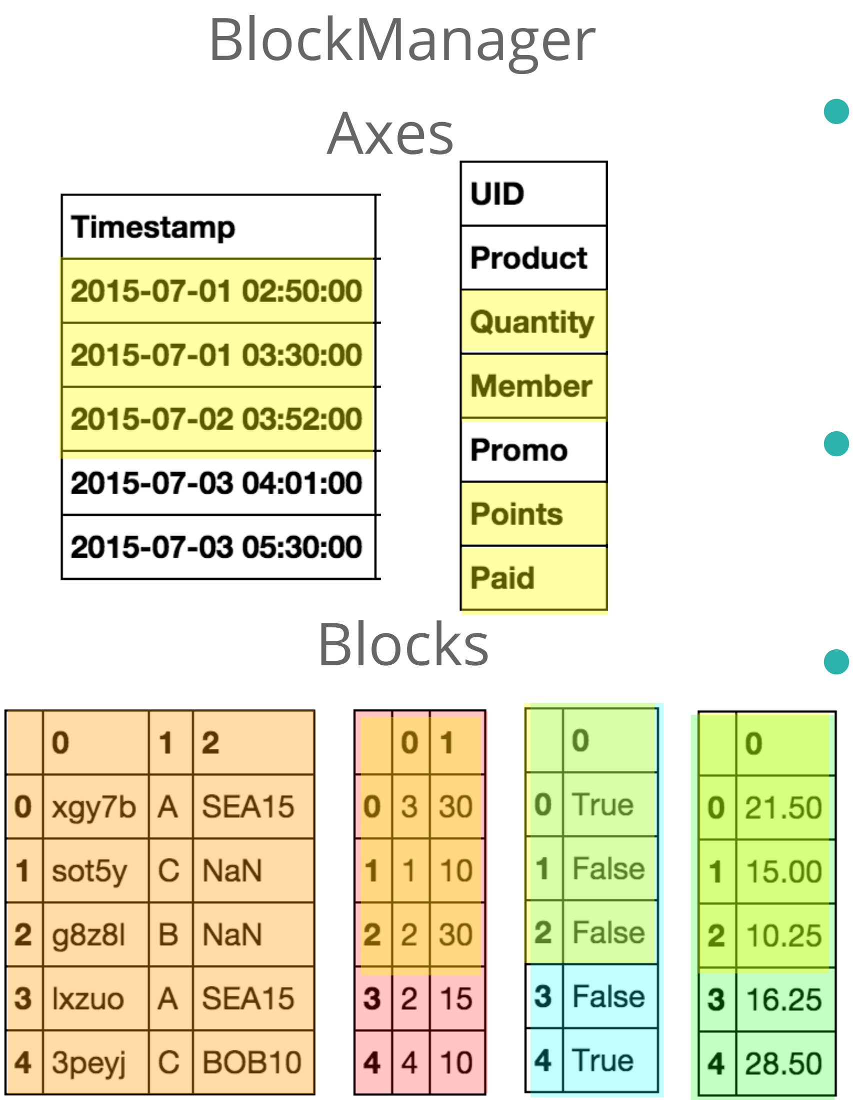
	0	1
0	3	30
1	1	10
2	2	30
3	2	15
4	4	10

	0
0	True
1	False
2	False
3	False
4	True

	0
0	21.50
1	15.00
2	10.25
3	16.25
4	28.50

- 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

BlockManager

Axes

Timestamp
2015-07-01 02:50:00
2015-07-01 03:30:00
2015-07-02 03:52:00
2015-07-03 04:01:00
2015-07-03 05:30:00

UID
Product
Quantity
Member
Promo
Points
Paid

Blocks

	0	1	2
0	xgy7b	A	SEA15
1	sot5y	C	NaN
2	g8z8l	B	NaN
3	lxzuo	A	SEA15
4	3peyj	C	BOB10

	0	1
0	3	30
1	1	10
2	2	30
3	2	15
4	4	10

	0
0	True
1	False
2	False
3	False
4	True

	0
0	21.50
1	15.00
2	10.25
3	16.25
4	28.50

- 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

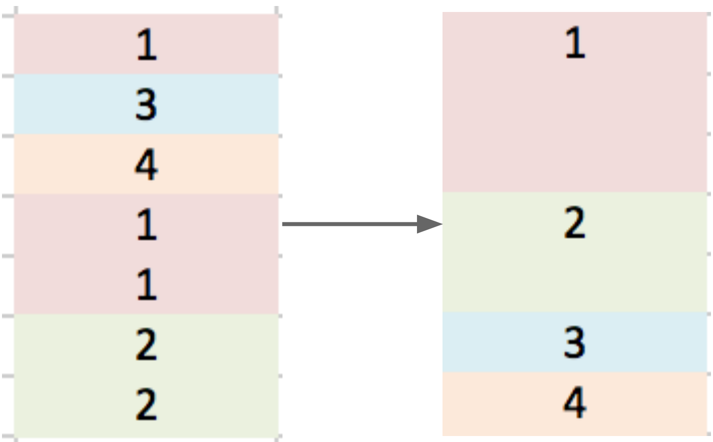
Summary

Factorizing underlies key pandas ops

```
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])
```

- 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

Motivation: Counting Sort (or “group sort”)

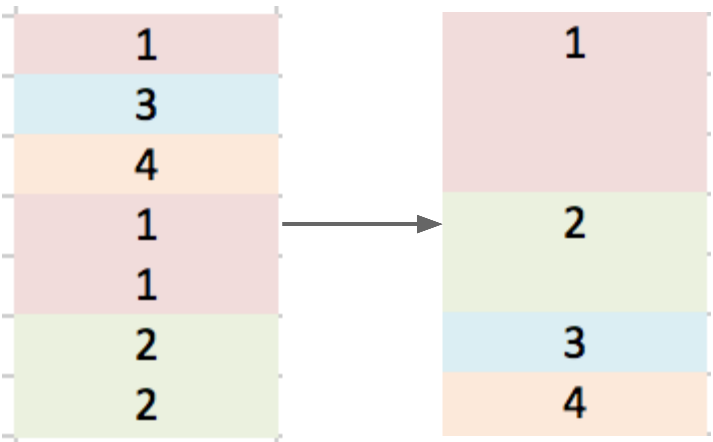


- Imagine you have 100k rows, but only 10k unique values
- Instead of comparisons (O(NlogN)), can scan through, grab unique values and the *count* of how many times each value occurs
- now you know bin size and bin order

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

Counsyl