Pandas `.head()` to `.tail()`

Tom Augspurger

Introduction

Background Expectations

- Hopefully you've used Python before
- Experience with NumPy will be helpful, but not required
- Pandas will be the primary focus
- We'll see bits of scikit-learn and statsmodels

Course Format

- I’ll have slides
- We’ll work through notebooks (execute each cell)
- The **slide title** will match the **notebook section**
- You’ll do exercises
- During exercises, I’ll follow-up on submitted questions
- I’ll demonstrate the solutions

The Jupyter Notebook is a web application that allows you to create and share documents that contain live code, equations, visualizations and explanatory text.

- Two Modes: Edit and Command
- Command -> Edit: `Enter`
- Edit -> Command: `Esc`
- Execute a Cell: `Shift+Enter`
- Down: `j/Down Arrow`
- Up: `k/Up Arrow`

Tab Completion

IPython will tab complete method names and function arguments

Use `shift+tab` to inside a function call to show the signature

Exercises

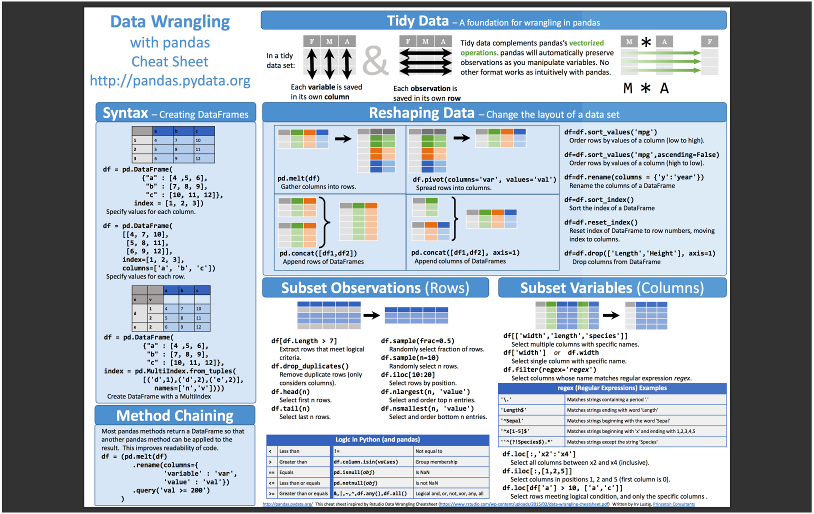
- Lots of small exercises to check understanding
- Each exercise includes
    - A prompt / question to be answered
    - An empty cell for code
    - A “magic” cell that loads a solution
- Execute the magic cell twice

Exercise 1 Print ’Hello, world!’

Print the text “Hello, world!”

Pandas Cheat Sheet

https://github.com/pandas-dev/pandas/blob/master/doc/cheatsheet/Pandas_Cheat_Sheet.pdf



Data Wrangling with pandas Cheat Sheet

http://pandas.pydata.org

Tidy Data – A foundation for wrangling in pandas

In a tidy data set: Each **variable** is saved in its own **column** & Each **observation** is saved in its own **row**

Tidy data complements pandas's vectorized operations. pandas will automatically preserve observations as you manipulate variables. No other format works as intuitively with pandas.

M * A

Syntax – Creating DataFrames

	a	b	c
1	4	7	10
2	5	8	11
3	6	9	12

```
df = pd.DataFrame(
        {"a" : [4 ,5, 6],
         "b" : [7, 8, 9],
         "c" : [10, 11, 12]},
       index = [1, 2, 3])
```

Specify values for each column.

```
df = pd.DataFrame(
     [[4, 7, 10],
      [5, 8, 11],
      [6, 9, 12]],
     index=[1, 2, 3],
     columns=['a', 'b', 'c'])
```

Specify values for each row.

n	v	a	b	c
d	1	4	7	10
	2	5	8	11
e	2	6	9	12

```
df = pd.DataFrame(
        {"a" : [4 ,5, 6],
         "b" : [7, 8, 9],
         "c" : [10, 11, 12]},
index = pd.MultiIndex.from_tuples(
        [('d',1),('d',2),('e',2)],
          names=['n','v']))
```

Create DataFrame with a MultiIndex

Method Chaining

Most pandas methods return a DataFrame so that another pandas method can be applied to the result. This improves readability of code.

```
df = (pd.melt(df)
        .rename(columns={
               'variable' : 'var',
               'value' : 'val'})
        .query('val >= 200')
     )
```

Reshaping Data – Change the layout of a data set

`pd.melt(df)`
Gather columns into rows.

`df.pivot(columns='var', values='val')`
Spread rows into columns.

`pd.concat([df1,df2])`
Append rows of DataFrames

`pd.concat([df1,df2], axis=1)`
Append columns of DataFrames

`df=df.sort_values('mpg')`
Order rows by values of a column (low to high).

`df=df.sort_values('mpg',ascending=False)`
Order rows by values of a column (high to low).

`df=df.rename(columns = {'y':'year'})`
Rename the columns of a DataFrame

`df=df.sort_index()`
Sort the index of a DataFrame

`df=df.reset_index()`
Reset index of DataFrame to row numbers, moving index to columns.

`df=df.drop(['Length','Height'], axis=1)`
Drop columns from DataFrame

Subset Observations (Rows)

`df[df.Length > 7]`
Extract rows that meet logical criteria.

`df.drop_duplicates()`
Remove duplicate rows (only considers columns).

`df.head(n)`
Select first n rows.

`df.tail(n)`
Select last n rows.

`df.sample(frac=0.5)`
Randomly select fraction of rows.

`df.sample(n=10)`
Randomly select n rows.

`df.iloc[10:20]`
Select rows by position.

`df.nlargest(n, 'value')`
Select and order top n entries.

`df.nsmallest(n, 'value')`
Select and order bottom n entries.

Logic in Python (and pandas)			
<	Less than	!=	Not equal to
>	Greater than	df.column.isin(*values*)	Group membership
==	Equals	pd.isnull(*obj*)	Is NaN
<=	Less than or equals	pd.notnull(*obj*)	Is not NaN
>=	Greater than or equals	&,|,~,^,df.any(),df.all()	Logical and, or, not, xor, any, all

Subset Variables (Columns)

`df[['width','length','species']]`
Select multiple columns with specific names.

`df['width']` *or* `df.width`
Select single column with specific name.

`df.filter(regex='regex')`
Select columns whose name matches regular expression *regex*.

regex (Regular Expressions) Examples	
'\.'	Matches strings containing a period '.'
'Length$'	Matches strings ending with word 'Length'
'^Sepal'	Matches strings beginning with the word 'Sepal'
'^x[1-5]$'	Matches strings beginning with 'x' and ending with 1,2,3,4,5
''^(?!Species$).*'	Matches strings except the string 'Species'

`df.loc[:,'x2':'x4']`
Select all columns between x2 and x4 (inclusive).

`df.iloc[:,[1,2,5]]`
Select columns in positions 1, 2 and 5 (first column is 0).

`df.loc[df['a'] > 10, ['a','c']]`
Select rows meeting logical condition, and only the specific columns .

http://pandas.pydata.org/ This cheat sheet inspired by Rstudio Data Wrangling Cheatsheet (https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf) Written by Irv Lustig, Princeton Consultants

Notebooks

1. Indexing
2. Alignment
3. Iterators & Groupby
4. Visualization
5. Tidy Data
6. Performance
7. Timeseries
8. Ecosystem

Data Structures and Indexing

Reading Data

Pandas has support for reading from many data sources, including

- `pd.read_csv`
- `pd.read_excel`
- `pd.read_html`
- `pd.read_json`
- `pd.read_hdf`
- `pd.read_sql`

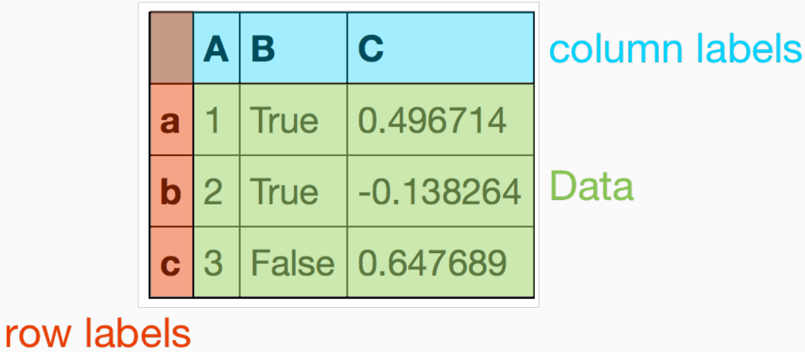


Figure 2: A dataframe is made up of data, row labels, and column labels

Data Types

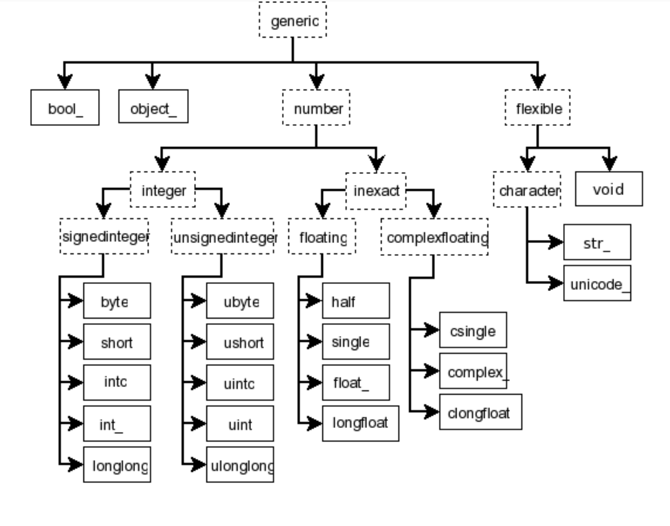


Figure 3:

Preview

A taste of where we’ll be by the end of the course

Goals of Indexing

There are many ways you might want to specify which subset you want to select:

- Like lists, you can index by integer position.
- Like dictionaries, you can index by label.
- Like NumPy arrays, you can index by boolean masks.
- You can index with a scalar, `slice`, or array
- Any of these should work on the index (row labels), or columns of a DataFrame, or both
- And any of these should work on hierarchical indexes.

1. Use `__getitem__` (square brackets) to select columns of a `DataFrame`

   ```
   >>> df[['a', 'b', 'c']]
   ```

2. Use `.loc` for label-based indexing (rows and columns)

   ```
   >>> df.loc[row_labels, column_labels]
   ```

3. Use `.iloc` for position-based indexing (rows and columns)

   ```
   >>> df.iloc[row_positions, column_positions]
   ```

The arguments to `.loc` and `.iloc` are `.loc[row_indexer, column_indexer]`. An indexer can be one of

- A scalar or array (of labels or integer positions)
- A `slice` object (including `:` for everything)
- A boolean mask

The column indexer is optional. We'll walk through all the combinations below.

Let’s select the two delay columns. Since we’re *only* filtering the columns (not rows), we can use dictionary-like `[]` to do the slicing.

Exercise 2 Select Columns by Name

Select the two airport-name columns, `'origin'` and `'dest'`, from `first`

Column . lookup

As a convenience, pandas attaches the column names to your `DataFrame` when they're valid python identifiers, and don't override one of the (many) methods on `DataFrame`

Label-Based Indexing with `.loc`

You can slice rows by label (and optionally the columns too) with `.loc`.
Let's select the rows for the carriers 'AA', 'DL', 'US', and 'WN'.

`slice` objects

You can pass a `slice` object (made with a `:`). They make sense when your index is sorted, which ours is.

Exercise 3 Index Rows and Columns

Select the columns `tail_num`, `origin`, and `dest` for the carriers `US`, `VX`, and `WN` from `first`.

Filter using a *1-dimensional* boolean array with the same length.

Exercise 4 Boolean Indexing

Select the rows of `flights` where the flight was cancelled (`cancelled == 1`)

Exercise 5 Boolean Indexing (2)

Filter down to rows where the departure **`hour`** is before 6:00 or after 18:00.

This takes the same basic forms as `.loc`, except you use integers to designate *positions* instead of labels.

What if you want all items *except* for some?

```
DataFrame.drop(labels, axis=0, ...)

Parameters
----------
labels : single label or list-like
axis : int or axis name
    - 0 / 'index', look in the index.
    - 1 / 'columns', look in the columns
```

Exercise 6 Dropping Row Labels

Use `first.drop` to select all the rows *except* `EV` and `F9`.

Exercise 7 Drop a column

`flights.airline_id` is redundent with `unique_carrier`. Drop `airline_id`.

Special Case: `DateTimeIndex`

Easier slicing with strings

Exercise 8 Datetime Indexing

Slice `delays` to select all rows from 12:00 on January 3rd, to 12:00 on the 10th.

Exercise 9 Thought Exercise

Why does pandas use a property like `.loc[..., ...]`, rather than a method like `.loc(..., ...)`?

- Introducted to `DataFrame` (2-D tabel) and `Series` (1-D array)
- Both have *row labels*, `DataFrame` also has `column labels`
- Saw `.loc` for labeled indexing and `.iloc` for positional indexing
- `.loc`, `.iloc`, and `__getitem__` all accept boolean masks too

Additional Exercises

Some additional exercises focused on indexing:

Alignment & Operatrions

Alignment

- Working with multiple pandas objects
- Strucuturing your data to make analysis easier
- Using labels to their full potential

Alignment without row labels (bad)

- separate datasets on GDP and CPI
- Goal: compute real GDP
- Problem: Different frequencies

- nomial GDP: Total output in dollars
- real GDP: Total output in constant dollars
- $\text{real gdp} = \frac{\text{nomial gdp}}{\text{inflation}}$

1. The output has lost the `DATE` fields, we would need to manually bring those along after doing the division
2. We had to worry about doing the merge, which is incidental to the problem of calculating real gdp

The Better Way

- Use row labels
- Specify `index_col='DATE'` in `read_csv`
- Just do the operation: `gdp / cpi`

Explicit Alignment

Roughly speaking, alignment composes two operations:

1. union the labels
2. reindex the data to conform to the unioned labels, inserting `NaN`s where necessary

Exercise 10 Compute Real GDP

Compute real GDP in 2009 dollars

Alignment on *both* axis

This may surpise you at some point down the road

Aside: Handling Missing Data

Pandas, recognizing that missing data is a fact of life, has a bunch of methods for detecting and handling missing data.

1. detecting missing data
2. dropping missing data
3. filling missing data

Detecting Missing Data

You can drop missing values with `.dropna`

```
DataFrame.dropna

Return object with labels on given axis omitted where
alternately any or all of the data are missing

Parameters
----------
axis : {0 or 'index', 1 or 'columns'}, or tuple/list thereof
    Pass tuple or list to drop on multiple axes
how : {'any', 'all'}
   * any : if any NA values are present, drop that label
    * all : if all values are NA, drop that label
```

Dropna for DataFrames

Since `DataFrame` is a 2-d container, there are additional complexities with dropping missing data. Do you drop the row or column? Does just one value in the row or column have to be missing, or all of them?

Exercise 11 Dropping Columns

Drop any `columns` in `df` that have at least one missing value

Use `.fillna` to fill with a value (scalar, or mapping of `label: value`) or method.

Joining Pandas Objects

You have some options:

1. `pd.merge`: SQL-style joins
2. `pd.concat`: array-style joins

Exercise 12 Merge Datasets

Use `pd.merge` to join the two DataFrames `gdp_bad` and `cpi_bad`, using an *outer* join (earlier we used an *inner* join).

Exercise 13 Concatenate Datasets

Use `pd.concat` to stick together `gdp` and `cpi` into a DataFrame

These next couple of topics aren't really related to alignment, but I didn't have anywhere else to put them.

NumPy has the concept of universal functions (ufuncs) that operate on any sized array.

`DataFrame` has many methods that *reduce* a DataFrame to a Series by aggregating over a dimension. Likewise, `Series` has many methods that collapse down to a scalar. Some examples are `.mean`, `.std`, `.max`, `.any`, `.all`.

Exercise 14 Percent Positive

Exercise: What percent of the periods had a positive percent change for each column?

Exercise 15 JOLTS

(This is an optional exercise, if you're working ahead).

During the housing bubble and financial crisis, CalculatedRisk was one of the best places for information on the internet. Let's reproduce one of his charts:

- Auto-alignment in pandas is different than most other systems
- Let pandas handle the details of alignment, you worry about important things
- Pandas methods are non-mutating
- `.dropna`, `.filla`, `isnull` for handling missing data

Iterators

Topics

- Stream larger-than-memory data through a pipeline
- Composable thanks to the iterator protocol
- Relatively easy to read and write

- A review is a list of lines
- Each review line is formated like `meta/field: value`
- Reviews are separated by blank lines (i.e. the line is just `'\n'`)

Let’s build a solution together. I’ll provide some guidance as we go along.

Parsing Tasks

1. split the raw text stream into individual reviews
2. transform each individual review into a data container
3. combine a chunk of transformed individual reviews into a collection
4. store the chunk to disk

Exercise 16 Format Review

Write a function `format_review` that converts an item like `first` into a dict

To a DataFrame

Assuming we've processed many reviews into a list, we'll then build up a DataFrame.

Full pipeline

1. `file -> review_lines : List[str]`
2. `review_lines -> reviews : Dict[str, str]`
3. `reviews -> DataFrames`
4. `DataFrames -> CSV`

Breief Aside on Dask

Dask is a flexible parallel computing library for analytic computing.

I’ve provided the reviews by the top 100 reviewers. We’ll use it for talking about groupby.

Pandas has been expanding its use of namespaces (or accessors) on `DataFrame` to group together related methods. This also limits the number of methods direclty attached to `DataFrame` itself, which can be overwhelming.

Currently, we have these namespaces:

- `.str`: defined on `Series` and `Index`es containing strings (object dtype)
- `.dt`: defined on `Series` with `datetime` or `timedelta` dtype
- `.cat`: defined on `Series` and `Indexes` with `category` dtype
- `.plot`: defined on `Series` and `DataFrames`

Exercise 17 Reviews by Hour

Make a barplot of the count of reviews by hour of the day.

Exercise 18 Pale Ales

Make a variable `pale_ales` that filters `df` to just rows where `beer_style` contains the string `'pale ale'` (ignoring case)

Groupby

Components of a groupby

1. **split** a table into groups
2. **apply** a function to each group
3. **combine** the results into a single DataFrame or Series

Split

Break a table into smaller logical tables according to some rule

Apply & Combine

To finish the groupby, we apply a method to the groupby object.

Exercise 19 Highest Variance

Find the `beer_style`s with the greatest variance in `abv`.

.agg output shape

The output shape is determined by the grouper, data, and aggregation

- Grouper: Controls the output index
    - single grouper -> Index
    - array-like grouper -> MultiIndex
- Subject (Groupee): Controls the output data values
    - single column -> Series (or DataFrame if multiple aggregations)
    - multiple columns -> DataFrame
- Aggregation: Controls the output columns
    - single aggfunc -> Index in the colums
    - multiple aggfuncs -> MultiIndex in the columns (Or 1-D Index if groupee is 1-D)

Exercise 20 Rating by length

Plot the relationship between review length (number of characters) and average `reveiw_overall`.

Exercise 21 Reviews by Length

Find the relationship between review length (number of **words** and average `reveiw_overall`.)

Exercise 22 Rating by number of Reviews

Find the relationship between the number of reviews for a beer and the average `review_overall`.

A *transform* is a function whose output is the same shape as the input.

Exercise 23 Personal Trend?

Do reviewer's `review_overall` trend over a person's time reviewing?

General `.apply`

We've seen `.agg` for outputting 1 row per group, and `.transform` for outputting 1 row per input row.

The final kind of function application is `.apply`. This can do pretty much whatever you want. We'll see an example in a later notebook.

Summary

- We used Python's iterator protocol to transform the raw data to a table
- We saw how Dask could handle larger-than-memory data with a familiar API
- We used groupby to analyze data by subsets

Visualization

Matplotlib

- foundation for seaborn and pandas plotting
- full control over every detail

Usually convenient

- Previously, nicer aesthetics (not since matplotlib 2.0)
- Nicer labeling (but matplotlib is better now)
- Easier (though less flexible) subplotting

Seaborn provides a high-level interface for drawing attractive statistical graphics.

- Statistical aggregations (`countplot`, bootstrapped standard errors, `regplot`)
- Easier distribution plotting
- Easier faceting by variable

Titanic Dataset

- Survived
- Class
- Sex
- Age
- Embarked
- Man / Woman / Child
- Deck

1. Who were the passengers?
2. Who survived?

Who were the passengers?

Explore them across different dimensions; We'll start with *categorical* data like sex or class.

What's the count of passengers by sex?

Exercise 24 Embarked by class

Make a `factorplot` with the counts of `embarked`, with the `hue` split by `class`.

Exercise 25 Age by class

Make a pointplot of `age` by `class`. Look at the `kind` parameter to `sns.factorplot`.

Let's moving to plotting *quantitative* data. We'll do this while introducting a new abstraction from seaborn, the `Grid` (`Grid`s work with either quantitative or qualitative data).

You initalize a `Grid` with all the agruments needed to layout the grid that the data will be plotted on:

- `data`: DataFrame
- `row` : variable to facet rows by
- `col` : variable to facet columns by
- `hue` : variable to split colors by

Exercise 26 Trimming

Create a new column in `t` called `fare_` that topcodes `fare` to be no more than `3 * t.fare.median()`. That is, anything higher than 3x the median should just be set to 3x the median.

We've seen summary statistics (like countplot), univariate distributions, and basic relationships between one variable and a categorical variable.

Seaborn also provides tools for visualizng bivariate relationships between quantitative variables.

Who Survived?

Let's turn to the variable of interest: who survived?

Exercise 27 Who Survived?

Explore the `alive` variable

Regression plots

You can plot relationships with best fit lines (and bootstrapped standard errors) using `lmplot`.

Exercise 28 Survived by gender

Can you split that relationship by `sex`?

- Many small functions with a consistent API (`x`, `y`, `data`, etc.)
- `Grid`s offer an abstraction for (relatively) easy faceting

Altair

Tidy Data

Structuring datasets to facilitate analysis (Wickham 2014)

In a tidy dataset...

1. Each variable forms a column
2. Each observation forms a row

Earlier, I fetched some data

```
tables = pd.read_html(
  "http://www.basketball-reference.com/leagues/"
  "NBA_2016_games.html"
)
games = tables[0]
games.to_csv('data/games.csv', index=False)
```

The Question:

How many days of rest did each team get between each game?

Melt

- Collect a variable spread across multiple columns into one, but
- Repeat the metadata to stay with each observation

pivot_table

You can “invert” a `melt` with `pd.pivot_table`

Two datasets

- `tidy`: For team-level questions
- `df`: For game-level questions

Exercise 29 Win Percent

Find the win-percent for each team, by whether they're home or away.

Stack / Unstack

- stack: `DataFrame` -> `Series` with `MultiIndex`
- unstack: `Series` with `MultiIndex` -> `DataFrame`

Exercise 30 Home Court Advantage?

How much of home court advantage can be explained by rest?

Modify `df` to include a couple potential targets

- `home_win`: binary indicator for whether the home team won
- `point_spread`: the home score minus the away score

Step 1: Team Strength

Most examples I've seen use a "team strength" variable in their regression estimating the home court advantage. We'll grab one from ESPN.

Create a new column `rest_spread` that contains the difference in rest (home - away)

Let's do some checks to see if we're on the right track. Does the home team typically have more rest?

Step 4: Regression

Now we can fit the model using statsmodels

Recap

- Tidy data:
    - one variable per column
    - one row per observation
- Methods:
    - melt / stack: wide to long
    - pivot_table / unstack: long to wide

Performance

With pandas, you'll get the most bang for your buck by *avoiding antipatterns*. There are additional options like using Numba or Cython if you *really* need to optimize a piece of code, but that's more work typically.

Mistake 1: Using pandas

- At least not for things it's not meant for.
- Pandas is very fast at joins, reindex, factorization
- Not as great at, say, matrix multiplications or problems that aren't vectorizable

Mistake 2: Using object dtype

Avoid it if possible

Aside: Managing Dtypes

Pandas provides some tools for converting arrays to their specialized dtype.

0. IO operations (`read_csv` infers, but can use the `dtype` keyword)
1. Object -> numeric: `pd.to_numeric`
2. Object -> datetime: `pd.to_datetime`
3. Object -> timedelta: `pd.to_timedelta`
4. Object -> category: `pd.Categorical`
5. `.astype(dtype)`

Pandas has a custom datatype, `Categorical`, for representing data that can come from a specified, generally fixed set of values.

- `categories`: set of valid values
- `ordered`: whether that set of values has an ordering

Internally, this is a dictionary encoding. The set of categories are stored *once*. The values `['a', 'b', 'c', 'a']` are stored as an array of integers, called `codes`.

When your collecting many different sources (say a bunch of separate CSVs) into a single DataFrame, you have two paths to the same goal:

1. Make a single empty DataFrame, append to that
2. Make a list of many DataFrames, concat at end

Typically, in python we'd choose the first one if we were, for example, collecting things into a `list`. `list.append` is very fast. However `DataFrame.append` is *not* fast.

Mistake 4: Doing too much work

This is more general purpose advice, rather than something you can just grep your code for. But look for places where you're doing a bunch of work, and then throwing some of it away.

Exercise 31 Nearest Neighbor

Find the nearest neighbor for all the airports with at least 500 departures.

Mistake 5: Using `.apply` (with axis=1) (Avoid Iteration)

I see this one a lot. I don't like absolutes, but you should never use `.apply(..., axis=1)` (probably). The root problem is using for loops instead of a vectorized solution. That is, something like:

I/O

Timeseries

- `pd.Timestamp` (nanosecond resolution `datetime.datetime`)
- `pd.Timedelta` (nanosecond resolution `datetime.timedelta`)

Resampling is similar to a groupby, but specialized for datetimes. Instead of specifying a column of values to group by, you specify a `rule`: the desired output frequency. The original data is binned into each group created by your rule.

Exercise 32 Resample

Plot the standard deviation for the number of flights from `MDW` and `ORD` at a weekly frequency

Exercise 33 Resample-Agg

Compute the the total number of flights (sum), mean, and median flights *per Quarter*.

Rolling, Expanding

Applying functions to windows, moving through your data.

pandas can store an array of datetimes with a common timezone. Right now the index for `df` is timezone naïve, but we can convert to a timezone with `tz_convert`:

I wish the standard library `datetime` module had something like this. Let's generate some fake data with `pd.date_range`

Being able to add columns of dates and timedeltas turns out to be quite convenient. Let's go all the way back to our first example with flight delays from New York airports.

Exercise 34 Convert Timedelta

Convert `flights.dep_delay` and `flights.arr_delay` to timedelta dtype.

Exercise 35 Timedelta Math

Compute the actual time the flight left, but adding the departure time `dep` and the delay `dep_delay`.

Modeling Timeseries

AutoRegressive Model

Predict y_{t+1}, given $y_0, y_1, \ldots y_t$

The real value of timeseries analysis is to predict the future. We can use the `.get_prediction` method to get the predicted values, along with a confidence interval.

Further Resources

- statsmodels state space docs
- statsmodels state space examples
- pyflux, another time series modeling library
- Sean Abu's post on ARIMA
- Jeffrey Yau's talks at PyData
- My blog post

Pandas and Scikit-Learn

It's the goto library for machine learning in Python. They use a consistent API for specifiying and fitting models. For *supervised* learning tasks, you have a *feature matrix* `X`, that's an `[N x P]` NumPy array, and a *target array* `y`, that's typically a 1-dimensional array with length `N`.

The Problem

1. Different data models:
    - NumPy is homogenous, n-dimensional arrays
    - Pandas is heterogenous, 2-dimensional tables
2. Pandas has additional dtypes

NumPy `ndarray`s (and so scikit-learn feature matrices) are *homogeneous*, they must have a single dtype, regardless of the number of dimensions. Pandas `DataFrame`s are potentially *heterogenous*, and can store columns of multiple dtypes within a single DataFrame.

Pandas has implemented some *extension dtypes*: `Categoricals` and datetimes with timezones.

For our example we'll work with a simple dataset on tips:

Exercise 36 Target, Feature arrays

Split the DataFrame `df` into a `Series` called `y` containing the `tip` amount, and a DataFrame `X` containing everything else.

Our target variable is the tip amount. The remainder of the columns make up our features.

Our focus is about how to use pandas and scikit-learn together, not how to build the best tip-predicting model. To keep things simple, we'll fit a linear regression to predict `tip`, rather than some more complicated model.

Dummy Encoding

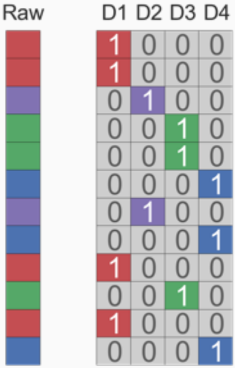


Figure 4: dummy

Our last approach worked, but there's still room for improvement.

1. We can't easily go from dummies back to categoricals
2. Doesn't integrate with scikit-learn `Pipeline` objects.
3. If working with a larger dataset and `partial_fit`, codes could be missing from subsets of the data.
4. Memory inefficient if there are many records relative to distinct categories

Aside: scikit-learn Pipelines

Pandas `Categorical` dtype

We've already talked about Categoricals, but as a refresher:

- There are a fixed set of possible values the variable can take
- The cateogories can be ordered or unordered
- The array of data is dictionary encoded, so the set of possible values is stored once, and the array of actual values is stored efficiently as an array of integers

We now have the pieces in place to solve all our issues. We'll write a class `DummyEncoder` for use in a scikit-learn `Pipeline`. The entirety is given in the next cell, but we'll break it apart piece by piece.

`.transform`

The transform method is the simplest, it's using `pd.get_dummies` like we did before. That is wrapped in a `np.asarray` to convert the DataFrame to a NumPy array, simulating what would happen if we pass the dummy-encoded class to a scikit-learn transformer that deals with NumPy arrays.

`.fit`

In `.fit`, we need to store all the information needed to take the `trn` NumPy array and go back to a `DataFrame` in the `.inverse_transform` step. This includes

- Column names (`self.columns_`)
- Cateogrical information (`self.cat_map_`)
- Mapping original columns to transformed positions (`self.non_cat_cols_` and `self.cat_blocks_`)

The first thing to realize is that pandas `get_dummies` returns the un-touched (numeric) columns first. We had two of those, `total_bill` and `size`, and collect those first in `inverse_transform`.

The rest of `inverse_transform` deals with `categoricals`. We have two separate tasks here.

1. Know which of the expanded columns in `trn` belong to which original categorical columns
2. Know the categorical attributes (`ordered`, `categories`) for that categorical

For the first task, we use the information stored in `self.cat_blocks_`. This is a dictonary mapping categorical column names to `slice` objects, that can be used on `trn`.

Using our pipeline

We explored some of the differences between the scikit-learn (NumPy) and pandas data models. We needed to convert a heterogenous pandas `DataFrame` to a homogonous `ndarray` for use in scikit-learn. Specifically we used `pd.get_dummies` to dummy encode the categorical data. After dummy encoding, we sucessfully fit the model.

For a more robust method, we implmented two scikit-learn `Pipeline` compatible transformers. The first converted columns of strings into proper pandas `Categorical`s. The second used `pd.get_dummies` to transform `Categoricals`, storing all the information needed to reverse the transformation.