PPOL 564: Data Science I

Unit 02: Pandas data wrangling

Roadmap

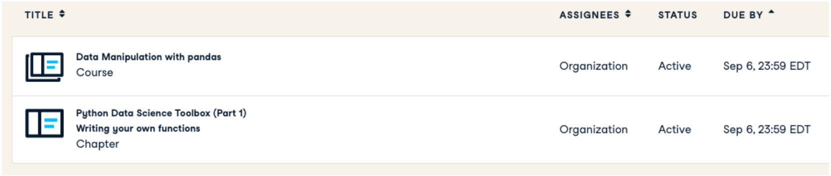
- Upcoming assignments
- Data wrangling using `pandas`
  - Intro to dataframes
  - Aggregation
  - Recoding data using logical conditions
  - Row and column filtering

Where we are

- **Upcoming assignments**
- Data wrangling using `pandas`
  - Intro to dataframes
  - Aggregation
  - Recoding data using logical conditions
  - Row and column filtering

Where we're headed

- **Friday 09/02 at 11:59 PM:** problem set one due
- **Saturday 09/03:** release problem set two; assigned groups of 2-3 we'll release before or with the pset
- **Tuesday 09/06:** DataCamps relevant to problem set two. Data manipulation with pandas module reiterates this lecture's material + intro to visualization; writing your own functions is relevant for next week's lecture

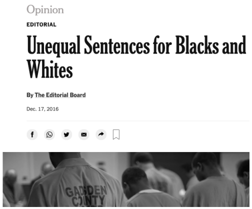


TITLE	ASSIGNEES	STATUS	DUE BY
Data Manipulation with pandas Course	Organization	Active	Sep 6, 23:59 EDT
Python Data Science Toolbox (Part 1) Writing your own functions Chapter	Organization	Active	Sep 6, 23:59 EDT

- **Wednesday 09/07 class:** overview of final project structure; user-defined functions; basic visualizations using `plotnine` (wrapper for ggplot2)
- **Friday 09/16:** problem set two due

Problem set two: policy background for sentencing disparities data

- **Data**: deidentified felony sentencing data from Cook County State's Attorney's Office (SAO)
- Released as part of push towards transparency with election of a new prosecutor in 2016



Opinion

EDITORIAL

Unequal Sentences for Blacks and Whites

By The Editorial Board

Dec. 17, 2016

Earlier this month Kim Foxx, the state's attorney for Cook County, Illinois, which covers Chicago, released six years' worth of raw data regarding felony prosecutions in her office. It was a simple yet profound act of good governance, and one that is all too rare among the nation's elected prosecutors. Foxx asserted that "for too long, the work of the criminal justice system has been largely a mystery. That lack of openness undermines the legitimacy of the criminal justice system." Source

Concepts in question 1

Problem set question	**Concepts**
1.1: unit of analysis in the data	Aggregating using groupby and agg
1.2.1 difference between original offense and updated offense	Creating new columns using np.where or checking equality
1.2.2 simplifying the charges	Pandas str.contains or list iteration
1.3: cleaning additional variables	Pandas str.contains (for race); np.where (for gender); Pandas quantile (for age); Pandas pd.to_datetime() (for converting a string column to a datetime column)
1.4: subsetting rows to analytic dataset	Row filtering

Concepts in question 2

Problem set question	Concepts
2.1 Over time variation in what % of cases are against Black defendants versus white defendants	Aggregating using groupby and agg
2.2 Over time variation in what % of Black versus white defendants face incarceration versus probation	Aggregating using groupby and agg
2.3.1 Common offenses	value_counts and sort values; set command
2.3.2 Between-group differences in incarceration for those common offenses	Aggregating using groupby and agg
2.3.3 Examining disparities before and after a policy change	Row filtering; user-defined function; list comprehension

Where we are

- Upcoming assignments
- **Data wrangling using `pandas`**
  - **Intro to dataframes**
  - Aggregation
  - Recoding data using logical conditions
  - Row and column filtering

Intro to data wrangling: how do dataframes differ from lists and arrays?

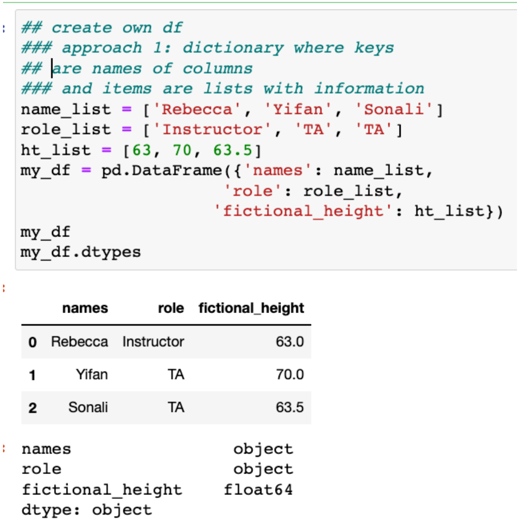
- In last lecture, we covered two structures for storing information in python:
  - Lists: structure built into python; 1-dimensional storage of information that can deal with information of different types in the same list
  - Arrays: requires the `numpy` package; n-dimensional (can be > 2) storage of information - usually use to store numeric information for efficient math calculations/model estimation
- DataFrames: 2-dimensional with rows (first dimension) and columns (second dimension) — sometimes called tabular data structure
  - `pandas` package (usually aliased as `pd`)
  - Each column can contain a different type of information
  - Each row references some unit of analysis (person; nation; city; 911 call; etc)

Two ways of interacting with dataframes

1. Creating our own
   - Less common
   - **Main use**: when we're creating data from some non-tabular source (e.g., a text string of a short politician bio; extract their name, party, and religious affiliation)
2. Reading in data stored in different formats
   - Focus for now: csv
   - Others we'll get to: excel; json; pkl or other serialized format; txt; spatial data stored as shapefiles

Creating our own dataframe: dictionary approach

Keys: column names; **values:** lists or arrays containing information

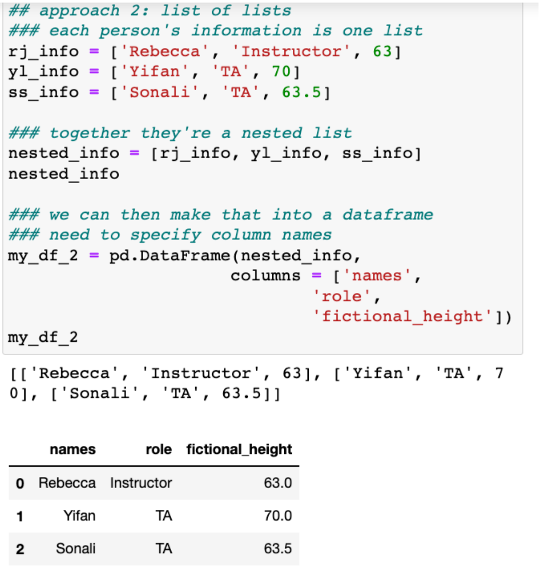


```
## create own df
### approach 1: dictionary where keys
## are names of columns
### and items are lists with information
name_list = ['Rebecca', 'Yifan', 'Sonali']
role_list = ['Instructor', 'TA', 'TA']
ht_list = [63, 70, 63.5]
my_df = pd.DataFrame({'names': name_list,
                     'role': role_list,
                    'fictional_height': ht_list})
my_df
my_df.dtypes
```

	names	role	fictional_height
0	Rebecca	Instructor	63.0
1	Yifan	TA	70.0
2	Sonali	TA	63.5

```
names                object
role                 object
fictional_height    float64
dtype: object
```

Creating our own dataframe: nested lists



```
## approach 2: list of lists
### each person's information is one list
rj_info = ['Rebecca', 'Instructor', 63]
yl_info = ['Yifan', 'TA', 70]
ss_info = ['Sonali', 'TA', 63.5]

### together they're a nested list
nested_info = [rj_info, yl_info, ss_info]
nested_info

### we can then make that into a dataframe
### need to specify column names
my_df_2 = pd.DataFrame(nested_info,
                      columns = ['names',
                                 'role',
                                 'fictional_height'])
my_df_2
```

```
[['Rebecca', 'Instructor', 63], ['Yifan', 'TA', 7
0], ['Sonali', 'TA', 63.5]]
```

	names	role	fictional_height
0	Rebecca	Instructor	63.0
1	Yifan	TA	70.0
2	Sonali	TA	63.5

More common way of interacting with dataframes: reading in data

- `os` package is important for finding the path of the file
  - `os.getcwd()` tells you the working directory you're in
- Two ways to structure path names (will return to these when we cover command line + GitHub in a couple weeks)
- **Way one (avoid if possible) absolute paths:**

  `'/Users/rebeccajohnson/Dropbox/ppol564_prepwork/prep_activities/f22_materials'`
- **Better way: relative paths to .py or .ipynb:** my data is stored two levels up from where my notebook is; can provide abbreviated pathname:

  `'../../data/example_data.csv'`
- Structure of read command pd.read_csv('path to file')

Once we have the data in Python, can summarize using built-in attributes or functions

- `dfname.dtypes:` returns a pandas series where the index is the name of the column; the value is the type of data it contains (eg str; int; float)
- `dfname.shape`: returns a length-2 tuple with dimensions of dataframe (number of rows, number of columns)
- `dfname.head()`: prints first n rows (defaults to 5)
- `dfname.tail()`: prints last n rows (defaults to 5)

Manipulations of data

Examples: finding mean height across the TAs; recoding heights into different categories; subsetting to a dataframe only containing TAs

names	**role**	**fictional_height**
Rebecca	Instructor	63.0
Yifan	TA	70.0
Sonali	TA	63.5

Where we are

- **Upcoming assignments**
- **Data wrangling using `pandas`**
  - Intro to dataframes
  - **Aggregation**
  - Recoding data using logical conditions
  - Row and column filtering

Aggregation syntax: one grouping variable and summarizing one column

```
grouping_result = df.groupby('grouping_varname').agg(
                  {'varname_imsummarizing': 'functiontosummarize'
                  }).reset_index()
```

- **Why is there a dictionary inside of agg?** Helps us tell it what to summarize by and which functions to use; keys are the variables to summarize by; values are what function to use
- **Why might we use reset_index()?** Just helps us treat the output as a dataframe with clear, one-level columns

Aggregation syntax: one grouping variable and summarizing multiple columns

```
grouping_result = df.groupby('grouping_varname').agg(
                 {'varname_imsummarizing': 'functiontosummarize',
                 'othervarname_imsummarizing': 'functiontosummarize'
                 }).reset_index()
```

- **When might this be useful?** If we have a boolean indicator–e.g., True/False that a person was incarcerated—we can group by a demographic category (e.g., race) and the mean of that boolean is the % yes in that category

Aggregation syntax: two grouping variables

```
grouping_result = df.groupby(['grouping_varname1',
                             'grouping_varname2']).agg(
                             {'varname_imsummarizing': 'functiontosummarize'
                             }).reset_index()
```

- **When might this be useful?** things like “how does this vary by time and category x?”

How do we structure the function inside the aggregation?

Three common ways of calling the function:

1. Functions that operate on panda series, e.g.:
   ```
   df.groupby('month').agg({'offense': ['nunique', 'first']})
   ```
2. Functions from numpy (aliased here as np), e.g.
   ```
   df.groupby('month').agg({'offense': [np.mean, np.mean]})
   ```
3. “Lambda” functions we write ourself that take an argument
   ```
   df.groupby('month').agg({'offense':
                            lambda x: len(x.unique())})
   ```

Summarizing over multiple rows or columns (without aggregation)

```
mean_threecols =      df[["colA", "colB",
                      "colC"]].apply("mean",
                      axis = 0)
```

- Pandas apply function: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.apply.html
- `axis` argument tells us whether to apply the function over columns (axis 0) or rows (axis 1)
- Can see in activity how the output is structured

Pause for practice

Aggregation section (section 1) of `00_pandas_datacleaning`

Where we are

- **Upcoming assignments**
- **Data wrangling using `pandas`**
  - Intro to dataframes
  - Aggregation
  - **Recoding data using logical conditions**
  - Row and column filtering

First type of column creation: binary indicators

Two general approaches that are "vectorized," or they work across all rows automatically without you needing to do a for loop:

1. `np.where`: similar to `ifelse` in R; useful if there's only 1-2 True/False conditions; can be used in conjunction with things like `df.varname.str.contains(‘‘some pattern’’)` if the column is string/char type
2. `np.select`: similar to `case_when` in R; useful for when there's either (1) several True/False conditions or (2) you're coding one set of categories into a different set of categories (e.g., pset question asking you to code any offense with `Arson` in the string into a single arson category)

Different types of np.where

```

## indicator for after 2020 christmas or not (make sure to
## format date in same way)
df['is_after_christmas'] = np.where(df.nameofdatecol > "2020-12-25"
    ,                                                      True, False)

## indicator for whether month is in spring quarter (april, may,
    june)
df['is_spring_q'] = np.where(df.monthname.isin(["April",
                            "May", "June"]),
                            True, False)

## indicator for whether someone's name contains johnson
df['is_johnson'] = np.where(df.fullname.str.contains("Johnson"),
                            True, False)

## strip string of all instances of johnson
df['no_johnson'] = df.fullname.str.replace("Johnson", "")
```

Then, if we created binary indicator, can use for subsetting rows

```

## subset to after christmas
df_afterchristmas = df[df.is_after_christmas].copy()

## subset to after christmas AND spring quarter
## note parantheses around each
df_postc_spring = df[(df.is_after_christmas) &
                    (df.is_spring_q)].copy()

## subset to after christmas BUT NOT spring quarter
## note tilde ~ for negation
df_postc_notspring = df[(df.is_after_christmas) &
                    (~df.is_spring_q)].copy()
```

np.where is useful for single conditions, but what about multiple conditions?

- **Example**: code to fall q if September, October, November, or December; code to winter q if January, February, or March; code to spring q if April, May, or June; code to summer q if otherwise
- Gets pretty ugly if nested np.where

```

## quarter ind
df["quarter_type"] = np.where(df.monthname.isin(["Sept",
                        "Oct", "Nov",  "Dec"]), "fall_q",
                    np.where(df.monthname.isin(["Jan",
                    "Feb", "March"]), "winter_q",
                    np.where(df.monthname.isin(["April",
                    "May", "June"]), "spring_q", "summer_q")))
```

One approach: np.select

```

## step one: create a list of conditions/categories
## i can omit last category if i want or specify it
quarter_criteria = [df.monthname.isin(["Sept", "Oct",
                     "Nov", "Dec"]),
                     df.monthname.isin(["Jan", "Feb", "March"]),
                     df.monthname.isin(["April", "May", "June"])]

## step two: create a list of what to code each category to
quarter_codeto = ["fall_q", "winter_q", "spring_q"]

## step three: apply and add as a col
## note i can use default to set to the residual category
## and here that's a fixed value; could also retain
## original value in data by setting default to:
## df["monthname"] in this case
df["quarter_type"] = np.select(quarter_criteria,
                                quarter_codeto,
                                default = "summer_q")
```

A second approach: map and dictionary

```

## step one: create a dictionary where each key is a value
## I want to recode and each value is what I should recode to
quarter_dict = {"Jan": "winter_q",
                "Feb": "winter_q",
                "March": "winter_q",
                "April": "spring_q",
                "May": "spring_q",
                "June": "spring_q",
                "Sept": "fall_q",
                "Oct": "fall_q",
                "Nov": "fall_q",
                "Dec": "fall_q"}

## step two: map the original col to the new values
## using that dictionary
df["quarter_type"] = df.monthname.map(quarter_dict).fillna("
    summer_q")
```

Each was still tedious; are there ways to further simplify?

- Depending on the example, rather than enumerating all the conditions (e.g., `[‘‘April", "May", "June"]`), you can use list comprehension to recode more efficiently
- **Example**:
  - We have a dataframe column containing Georgetown courses
  - We want to create a sub-list of PPOL-prefix courses without using `df.coursename.str.contains`

Applying list comprehension to variable recoding

```

## pool of courses
all_courses = df.coursename.unique()

## subset to those that contain PPOL anywhere
only_ppol = [course for course in all_courses
             if "PPOL" in course]
```

- **for course in all_courses**: iterates over the list of courses
- **if condition**: tells it when to retain
- **course** at beginning: just return as is and don't do anything
- if we wanted to not only select but also strip out the course number, would do something like...

```
import re
only_ppol_nonum = [re.sub("[0-9]+", "", course)
                   for course in all_courses
                   if "PPOL" in course]
```

Pause for practice

Recoding section (section 2) of `00_pandas_datacleaning`

Where we are

- **Upcoming assignments**
- **Data wrangling using `pandas`**
  - Intro to dataframes
  - Aggregation
  - Recoding data using logical conditions
  - **Row and column filtering**

Row filtering: combining multiple conditions

```python
## select all rows where course name starts with ppol and semester
    is fall
f22_courses = df[(df.coursename.str.startswith("PPOL")) &
(df.courseyear.str.contains("2022"))].copy()
```

Two notes

- Using pandas built in methods (`startswith` and `str` accessor)- what would happen with latter if the variable was not a string?
- Extra parantheses when putting multiple conditions together (different from R)

A useful tool for row filtering before we learn regex: pandas built-in functions

- `contains` and `startswith` are functions built into `pandas` that help us work with string/character variables
- General syntax: nameofdata.nameofstringcol.str.nameoffunction(argument if relevent)
- Examples:
  - df.stringvar.str.upper() and lower()
  - df.stringvar.str.replace()
  - df.stringvar.str.split() — defaults to splitting on spaces; can feed it other delimiters like ;

Column filtering: combining with list comprehension

```
## subset to columns: "studentid" and columns with "2021" in the
    name (wide-format data)
grades_2021 = df[["studentid"] +
                 [col for col in
                 df.columns
                 if "2021" in col]].copy()
```

Notes:

- Use .copy() to tell python that we're assigning a copy of the original dataframe (`df`) to the new object `grades_2021`; otherwise, gives us `SettingwithCopy` warning; ambiguity about whether we want any further changes to: (1) only apply to the slice or (2) propagate back to original df (in applied contexts, almost always want 1)
- Put "studentid" in brackets because i'm combining two lists

Pause for practice

Filtering section (section 3) of `00_pandas_datacleaning`