QSS20: Modern Statistical Computing

Session 03: Catch Up

Goal for today's session

- Three ways of working with course material, pset extension poll results, and how to submit
- Mapping each question to Pandas concepts and practicing each one using `dc_crim_2020` data from `00_latex_output_examples_solutions`
  1. Aggregation
  2. Variable recoding
  3. Loops (what you've learned) versus functions
- Break
- Problem set work

Goal for today's session

- **Three ways of working with course material, pset extension poll results, and how to submit**
- Mapping each question to Pandas concepts and practicing each one using `dc_crim_2020` data from `00_latex_output_examples`
  1. Aggregation
  2. Variable recoding
  3. Loops (what you've learned) versus functions
- Break
- Problem set work

Way 1: write code locally and manually use jupyter hub to retrieve files

- If working locally, need to have Anaconda or a way to edit .ipynb files; can launch Anaconda using GUI (see slack instructions) or via terminal command: `jupyter notebook`
- Ideally have python 3.8+ installed but python 3.7 should also work if that's the only version compatible with your OS
- To get files without needing to do anything with the `qss20_slides_activities` github repo:
  1. Log onto jupyter hub
  2. Click the check mark next to a file
  3. Download and work with locally in your `qss20_s21_assignments` folder
- **Pros:** no memory limits; don't need to interact with GitHub other than to submit your problem set
- **Cons:** less GitHub learning

Way 2: write code locally and use GitHub to retrieve files

- Make a local copy of the `qss20_slides_activities` repo (either clone or can fork)
- `git pull` when you want new files
- Copy over files to your `qss20_s21_assignments` folder
- **Pros:** same as on last slide but get more practice with GitHub

Way 3: write code on jupyter hub

- All materials from the repo qss_slides_activities are available on jupyter hub in shared/qss20/. The nice thing is they auto-appear without you needing to git clone or git pull
- The big flag with all directories in shared are that they are **read only**, or you can't directly edit files while they're still in shared
- If in your home directory, way to copy one file over from shared/qss20:

```
    cp shared/qss20/problemsets/01_pset1/pset1_groupexerci
```

- If in your home directory, way to copy an entire folder over from shared/qss20:

```
    cp -r shared/qss20/problemsets/01_pset1/ qss20_mywork/
```

- **Pros:** avoid package installation issues; easier troubleshooting since we're working from same version of python
- **Cons:** need to monitor memory utilization in top-right corner

Poll results for classwide problem set extension

TBD

How to submit the problem set

- Get the following files into your qss20_s21_assignments repo:
  - **Everyone:** .ipynb and pdf version of the individual portion
  - **One person per group:** .ipynb and pdf version of the group portion
- Two ways to get them on GitHub:
  1. Preferred: go through the git add, git commit, and git push process we reviewed in Session 02
  2. If you don't feel comfortable with that, manually upload them using the "upload" feature on your repo's web platform
- When it's ready to grade, assign me an issue to grade and include a link to the ipynb version of the file

Goal for today's session

- Three ways of working with course material, pset extension poll results, and how to submit
- **Mapping each question to Pandas concepts and practicing each one using `dc_crim_2020` data from `00_latex_output_examples`**
  1. Aggregation
  2. Variable recoding
  3. Loops (what you've learned) versus functions
- Break
- Problem set work

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; loop

Concepts in question 3

Problem set question	**Concepts**
3.1 Filter to incarceration and construct a sentence length variable	Row filtering; loop or function
3.2 Narcotics disparities within the same judge	Row filtering; multiple ways to find which judges have at least 20 Black and White defendants, but pivot_table() is one; Aggregation using groupby and agg
3.3 Matched pairs	Row filtering; loop or function

Goal for today's session

- Three ways of working with course material, pset extension poll results, and how to submit
- Mapping each question to Pandas concepts and practicing each one using `dc_crim_2020` data from `00_latex_output_examples`
  1. **Aggregation**
  2. Variable recoding
  3. Loops (what you've learned) versus functions
- Break
- Problem set work

Review of 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

Review of 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?** For the pset questions on disparities, can do one summary of `is_black_derived` and another summary of `is_white_derived`

Review of 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, 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())})
   ```

Pause for whole-group practice

Aggregation section of 00_latex_output_examples

Goal for today's session

- Three ways of working with course material, pset extension poll results, and how to submit
- Mapping each question to Pandas concepts and practicing each one using `dc_crim_2020` data from `00_latex_output_examples`
  1. Aggregation
  2. **Variable recoding**
  3. Loops (what you've learned) versus functions
- Break
- Problem set work

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 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
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

```python

## 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

```python

## 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 a loop to subset a larger list to create that list more efficiently
- **Example**:
  - We have a column containing Dartmouth courses (e.g., QSS17, QSS20, ECON20, GOV10)
  - We want to pull out the QSS-prefix courses without using `df.coursename.str.contains`

Looping through a list

```python

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

## subset to those that contain QSS anywhere
only_qss = [course for course in all_courses
            if "QSS" 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...

```python
import re
only_qss_nonum = [re.sub("[0-9][0-9]", "", course)
                  for course in all_courses
                  if "QSS" in course]
```

Pause for whole-group practice

Recoding section of `00_latex_output_examples`

Goal for today's session

- Three ways of working with course material, pset extension poll results, and how to submit
- Mapping each question to Pandas concepts and practicing each one using `dc_crim_2020` data from `00_latex_output_examples`
  1. Aggregation
  2. Variable recoding
  3. **Loops (what you've learned) versus functions**
- Break
- Problem set work

Common task

- Do something repeatedly to something else (e.g., do something to every row in a dataset; transform every column)
- Within pandas, the built in methods like `df.mean()`, `df.col.str.contains()` etc only get us so far
- Often may want to iterate over rows, check for something or do something, and store the result
    - **Example:** pset question 3.3 on (1) focusing on each defendant, and (2) seeing if we can find any matches who are a different race but the same age and gender

To make more concrete: we have a couple example crime reports

CCN	WARD	OFFENSE	REPORT_DAT
20165648	6	MOTOR VEHICLE THEFT	2020/11/19 21:25:50+00
20123250	2	MOTOR VEHICLE THEFT	2020/08/29 01:00:25+00

For each of these two crimes, we want to see if there are any reported crimes with (1) same ward and (2) reported within 20 minutes of the first crime. We want to pull all those crimes and rank them by time proximity.

For both approaches, define crimes to look for and crimes to look within

```
## two examples
CCN_examples = ['20165648', '20123250']

## crimes to look 4 matches for
crimes_lookfor = dc_crim_2020.loc[dc_crim_2020.CCN.astype(str).isin
                 (CCN_examples),
                 ['CCN', 'WARD', 'OFFENSE', 'report_dt']].copy()

## crimes to look for matches within
other_crimes = dc_crim_2020[~dc_crim_2020.CCN.astype(str).isin(
                 CCN_examples)].copy()
```

Approach 1: loop through crimes

```
## create empty container to store results
store_matches = {}
## loop through two example crimes
for i in range(0, crimes_lookfor.shape[0]):
    ## extract row
    one_row = crimes_lookfor.iloc[i]

    ## first, subset to crimes in same ward
    same_wards = other_crimes[other_crimes.WARD == one_row.WARD].copy()

    ## second, with those same-ward crimes, filter to crimes within
    ## 20 minutes after focal crime
    cutoff = one_row.report_dt +  timedelta(minutes=20)
    same_wards_sametime = same_wards[
                    (same_wards.report_dt >=  one_row.report_dt) &
                    (same_wards.report_dt <= cutoff)].copy()

    ## third, store the results
    store_matches[str(one_row.CCN)] = same_wards_sametime

## finally, rowbind results into one df
all_matches = pd.concat(store_matches)
```

How might we transition this to a function?

General structure of a function

```

## what is the function called?
## and what are its inputs?
def do_something(search_for: pd.DataFrame,
                search_in: pd.DataFrame):

    ## here i'm doing things
    ## (similar to meat of a for loop)
    my_output = search_in[search_in.something ==
                        search_for.something].copy()
    ## and so on...

    ## here i'm returning things
    ## (similar to last part of a loop, though don't need
    ## a within-function container)
    return(my_output)
```

Approach 2: first define a function

```python
def proximate_crimes(search_for,
                     search_in):

    ## first, subset to crimes in same ward
    same_wards = search_in[search_in.WARD ==
                           search_for.WARD].copy()

    ## second, with those same-ward crimes, construct indicator
    ## for reported within 20 minutes
    cutoff = search_for.report_dt + timedelta(minutes=20)
    same_wards_sametime = same_wards[
                (same_wards.report_dt >= search_for.report_dt) &
                (same_wards.report_dt <= cutoff)].copy()

    ## add col for focal match
    same_wards_sametime['focal_crime'] = search_for.CCN

    ## return
    return(same_wards_sametime)
```

Approach 2: applying the function

```
## apply to a single crime
one_match = proximate_crimes(search_for = crimes_lookfor.iloc[0],
                             search_in = other_crimes)

## iterate over the crimes, apply, and rowbind results
all_matches_list = [proximate_crimes(search_for =
                    crimes_lookfor.iloc[i],
                    search_in = other_crimes)
                    for i in
                    range(0, crimes_lookfor.shape[0])]
all_matches = pd.concat(all_matches_list)
```

Next

Break for problem set work