QSS20: Modern Statistical Computing

Unit 03: Pandas wrap-up, user-defined functions

Goals for today's session

- Logistics: office hours, deadlines
- More on list comprehensions & dataframes
- Part three of previous lecture (row and column subsetting)
- User-defined functions
  - Lecture slides + example
  - Group activity
- Walk through notebook with plotting example code

Office hours by day of the week

- Monday: 2:15-3:15 PM (Prof. OH), 8-9 PM (peer tutoring)
- Tuesday: 1:30-2:30 (TA OH)
- Wednesday: 2:15-3:15 (Prof. OH), 9-10 PM (peer tutoring)
- Thursday: 7-8 PM (peer tutoring)
- Friday: none (for now)

Links & locations:

- Sign up for virtual/private OH with Prof. Haber, drop-ins welcome to 103 Silsby
- Sign up and zoom link for group tutoring with Ramsey Ash, for now in 152 Baker
- Zoom link for TA OH with Eunice Liu

Upcoming deadlines

- **Problem set one:** due this Sunday 09/25 by 11:59 PM (EST)
  - Available on Canvas & GitHub
  - Submit on Canvas
- **Final project overview:** review website materials by Wednesday
- **Final project survey:** due before class Monday 09/26 (will post in Piazza)
- **Problem set two:** due Sunday 10/09 at 11:59 PM
  - Submit via GitHub (will review on Wednesday)
- Four late days available for use across psets (let TA know if you're using a late day)

Final project overview

- **Work in groups of 3-4**, decide yourself or opt into partner pool
- **Components**:
  - Write scientific report
  - Publish well-documented repository
  - Give status update presentation
- **Options for final projects**:
  - Social Impact Practicum (SIP)
  - Cook County felony sentencing data
  - Senior thesis/independent ongoing project (**can** do this as group of one)
- More info on SIP on Wednesday
  - **Special visitor**: Ashley Doolittle, Associate Director of Dartmouth Center for Social Impact (SIP lead)
  - Will talk more about SIP data & options

Where we are

- Logistics: office hours, deadlines
- **More on list comprehensions & dataframes**
- Part three of previous lecture (row and column subsetting)
- User-defined functions
  - Lecture slides + example
  - Group activity
- Walk through notebook with plotting example code

Lists: how to create

```
## create a list
list_new = [9, 19, 1988]
list_existing = [my_birth_month, my_birth_day,
                 my_birth_year]

print(list_new)
print(list_existing)
print(type(list_existing))
print(len(list_existing))
```

```
[9, 19, 1988]
[9, 19, 1988]
<class 'list'>
3
```

Things to note:

- `list_new` I created from scratch; `list_existing` I combined the objects I created earlier in the code
- Either way, use `[` with commas separating list elements
- `len` is a built-in function in Python (doesn't require us to import a package) that works with lists in addition to other types of objects

Basic list comprehension

- **Goal:** iterate over list elements and do something:
  - Filter: select a subset of list elements based on some condition
  - Transform: modify the elements of the list
    - General: modifies each element in the same way
    - Conditional: modifies some elements in some way; others in a different way
- List comprehensions have similar applications to *for loops* (often these are interchangeable), but list comprehension has many advantages (faster, less memory-intensive)

Example task

Want to convert the list with the three birthday elements—`[9, 19, 1988]`—into a single string: "09-19-1988"

General transformation

```
## copy over list to give more informative name
bday_info = list(list_existing)
print(bday_info)

## convert each element to a string
bday_info_string = [str(num) for num in bday_info]
print(bday_info_string)
```

```
[9, 19, 1988]
['9', '19', '1988']
```

Breaking this down:

- `str(num)` is the step that's doing the transformation
- `for num in bday_info` iterates over each of the three elements in the `bday_info` list
- `num` is a totally arbitrary placeholder; we could use `i`, `el`, or whatever; key is that it's the same between the iteration and transformation

Conditional transformation

What if we want to not just convert each element to string, but add a 0 if the str is one-digit? (So pad the 9 with a 0 as it's converted to a string)?

Conditional transformation

```
## conditional transformation - add a 0 if only 1-char long
bday_info_string_pad = ['0' + num if len(num) == 1
                        else num
                        for num in bday_info_string]
print(bday_info_string_pad)
```

```
['09', '19', '1988']
```

Breaking this down:

- **What stayed the same?** The iterating through elements `for num in bday_info`
- **What changed?** We added a condition using `if` and `else`, and using the built-in `len()` function we covered earlier
  - If it's a 1-character string, it uses the + to paste the string `'0'` onto it
  - Otherwise, it keeps the string as is

Especially powerful when combined with regular expressions (regex) that we'll cover later

```
## example of regex to separate days versus months
### import module
import re
### month pattern is 01...09 or 11 or 12
month_pattern = r'0[1-9]|1[1-2]'
example_date_str = ['09', '30', '01', '12', '11', '19']
### keep element in list if element matches pattern
keep_months = [el for el in example_date_str
               if re.search(month_pattern, el)]
keep_months
```

```
['09', '01', '12', '11']
```

Intro to information structures for data wrangling

- 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 (more on this in future classes)
- 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)

Creating dataframes: Usually we read 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/jhaber/Dropbox/qss20_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')

Where we are

- Logistics: office hours, deadlines
- More on list comprehensions & dataframes
- **Part three of previous lecture (row and column subsetting)**
- User-defined functions
  - Lecture slides + example
  - Group activity
- Walk through notebook with plotting example code

Where we are

- Logistics: office hours, deadlines
- More on list comprehensions & dataframes
- Part three of previous lecture (row and column subsetting)
- **User-defined functions**
  - **Lecture slides + example**
  - Group activity
- Walk through notebook with plotting example code

Lambda functions: general syntax

Grouping context, single column:

```
df_name.groupby(groupingcol)
          .agg({focalcol:
          lambda rowgroup: operation(rowgroup)})
```

Non-grouping, multiple columns:

```
df_name[[col1, col2]]
          .apply(
          lambda row: operation(row))
```

In both cases, lambda works as a "function marker" to indicate where the custom row/row-group transformation begins.

Example of lambda function from material on aggregating data

Used a one-line function (lambda function) to sort offenses from most to least frequent and pull the most-frequent offense:

```
dc_crim_2020.groupby(['WARD',
          'SHIFT']).agg({'OFFENSE':
          lambda x: x.value_counts(sort = True,
          ascending = False).index[0]})
```

Lambda functions versus “normal” python functions

- **Lambda functions**: think of as *single-use, throwaway* functions — code works there but if we wanted to perform similar operation (eg find most frequent weapon used), would need to copy/paste that lambda function into different aggregation calls (not super scalable)
- **“Normal” python functions covered in DataCamp**: defined using the `def` command — helps us save time/make code more readable by avoiding repetitive code

Same example putting the code inside a function

```
def most_common(one_col: pd.Series):
    '''
    Function to return name of most common category
    Parameters:
        one_col (pd.Series): pandas series

    Returns:
        top (str): string with name of most frequent category
    '''

    ## sort values
    sorted_series = one_col.value_counts(sort = True, ascending =
    False)
    ## get top
    top = sorted_series.index[0]
    ## return
    return(top)

## execute
dc_crim_2020.groupby(['WARD',
            'SHIFT']).agg({'OFFENSE':
             lambda x: most_common(x)})
```

Three ingredients in a user-defined function

1. **Name of function and inputs**: name is arbitrary; multiple inputs are separated by commas (later, we'll cover setting inputs to default values)

   ```
   def most_common(one_col: pd.Series):
   ```

2. **Meat of function:** what the function does inside with the inputs

   ```
   ## sort values
   sorted_series = one_col.value_counts(sort = True,
   ascending = False)
   ## get top
   top = sorted_series.index[0]
   ```

3. **Return statement (if any):** returning one or more outputs; note that non-returned objects (eg in this example, the `sorted_series`) are discarded

   ```
   ## return
   return(top)
   ```

Building a function together

See first part of this notebook to follow along with the code:

02_functions_part1_blank.ipynb

Task

Write a function that takes in two arguments–a dataframe and an integer of a Ward number

- The function should subset to:
  - That ward
  - The ward immediately 'below' that ward (if focal ward is Ward 2, Ward 1)
  - The ward immediately 'above' that ward (if focal ward is Ward 2, Ward 3)
- Find the number of unique crime reports (unique CCN) in each ward
- Should print the name + number of crimes in the ward with the most unique crime reports of that comparison set (returns nothing)

Breaking down into steps

1. Get the **meat** of the function working outside the function with one example
2. Figure out what parts of that meat you want to generalize
3. Get that generalization working outside the function
4. Construct the function
5. Execute it on the one example and make sure it produces same output as step 1
6. Execute it on multiple examples

Meat of function with one example (ward 3)

```
## get list of wards + neighbors
neighbor_wards = [3 - 1, 3 + 1]
wards_touse = [3] + neighbor_wards

## then, use isin command to subset the data
## to those wards
df_focal = dc_crim_2020[dc_crim_2020.WARD.isin(wards_touse)].copy()

## then, use groupby to find unique
ward_ccn = df_focal.groupby('WARD')['CCN'].nunique().reset_index()

## finally, get the top one (multiple ways)
top_ward = ward_ccn.sort_values(by = 'CCN',
            ascending = False).head(1)

## print
print("Ward with most crime reports is WARD " + str(top_ward['WARD'].values[0]) +
      " with N reports: " + str(top_ward.CCN.values[0]))
```

Many things we could generalize

Focusing on bolded two (ward and dataframe name) but large list; depends on what we want to use function to do:

- **Ward we're focusing on (hard coded to 3)**
- **Name of data frame (hard coded to** `dc_crim_2020`
- Name of ward column (hard coded to WARD)
- Number of neighbors to look at (hard coded to 1 above and 1 below)
- Name of crime identifier column (hard coded to CCN)

Highlighting parts where ward and dataframe name are hard coded

```
## get list of wards + neighbors
neighbor_wards = [3 - 1, 3 + 1]
wards_touse = [3] + neighbor_wards

## then, use isin command to subset the data
## to those wards
df_focal = dc_crim_2020[dc_crim_2020.WARD.isin(wards_touse)].copy()

## then, use groupby to find unique
ward_ccn = df_focal.groupby('WARD')['CCN'].nunique().reset_index()

## finally, get the top one (multiple ways)
top_ward = ward_ccn.sort_values(by = 'CCN',
              ascending = False).head(1)
```

Replace hard-coded parts with placeholder

```
## get list of wards + neighbors
neighbor_wards = [focal_ward - 1, focal_ward + 1]
wards_touse = [focal_ward] + neighbor_wards

## then, use isin command to subset the data
## to those wards
df_focal = df[df.WARD.isin(wards_touse)].copy()

## then, use groupby to find unique
ward_ccn = df_focal.groupby('WARD')['CCN'].nunique().reset_index()

## finally, get the top one (multiple ways)
top_ward = ward_ccn.sort_values(by = 'CCN',
            ascending = False).head(1)
```

Can still test outside the function

```
## testing obj
focal_ward = 3
df = dc_crim_2020.copy()

## get list of wards + neighbors
neighbor_wards = [focal_ward - 1, focal_ward + 1]
wards_touse = [focal_ward] + neighbor_wards

## then, use isin command to subset the data
## to those wards
df_focal = df[df.WARD.isin(wards_touse)].copy()

## then, use groupby to find unique
ward_ccn = df_focal.groupby('WARD')['CCN'].nunique().reset_index()

## finally, get the top one (multiple ways)
top_ward = ward_ccn.sort_values(by = 'CCN',
              ascending = False).head(1)
```

Then, putting it all together for the function

(see notebook for documentation; omitted here on slide for space reasons)

```
def compare_wards(focal_ward: int, df: pd.DataFrame):

    ## get list of wards to use
    neighbor_wards = [focal_ward - 1, focal_ward + 1]
    wards_touse = [focal_ward] + neighbor_wards

    ## subset to those
    df_focal = df[df.WARD.isin(wards_touse)].copy()

    ## find crimes per ward
    ward_ccn = df_focal.groupby('WARD')['CCN'].nunique().
    reset_index()

    ## finally, get the top one
    top_ward = ward_ccn.sort_values(by = 'CCN', ascending = False).
    head(1)

    ## print
    print("Ward with most reports of neighbors is WARD " + \
          str(top_ward['WARD'].values[0]) +
     " with N reports: " + str(top_ward.CCN.values[0]))
```

Executing repeatedly: can combine with list comprehension

```

## repetitive execution
compare_wards(focal_ward = 3, df = dc_crim_2020)
compare_wards(focal_ward = 6, df = dc_crim_2020)

## using list comprehension
[compare_wards(focal_ward = i, df = dc_crim_2020)
    for i in [3, 6]]
```

Latter may be especially useful if the function returns something that we later want to combine

Where we are

- Logistics: office hours, deadlines
- More on list comprehensions & dataframes
- Part three of previous lecture (row and column subsetting)
- **User-defined functions**
  - Lecture slides + example
  - **Group activity**
- Walk through notebook with plotting example code

Break for group activity

We provide the "outside of function" code; you work to generalize this into a function and execute

Section 2 of this notebook: 02_functions_part1_blank.ipynb

Where we are

- Logistics: office hours, deadlines
- More on list comprehensions & dataframes
- Part three of previous lecture (row and column subsetting)
- User-defined functions
  - Lecture slides + example
  - Group activity
- **Walk through notebook with plotting example code**
  - Can use any plotting syntax for problem set — popular ones are matplotlib (covered by DataCamp last chapter of introduction to pandas); seaborn; plotnine
  - Notebook gives plotnine syntax