Introduction to Python for Social Science

Lecture 3 - Data Structures and Pandas II

Musashi Harukawa, DPIR

3rd Week Hilary 2020

Overview

Last Week

- Graph, Tree and Tabular Data Structures
- Representations of information
- Introduction to `pandas`

This Week

This week we learn more advanced methods for working with data:

- Functions
- `apply` and vectorization
- `GroupBy`: Split-apply-combine
- Combining dataframes: append, concat and merge
- Long- vs wide-form data; melting data

Methods and Theory

Functions

- A function is a mapping of two sets that relates each element of the first set to exactly one element of the second set.
  - Formally, a function f is a mapping of elements of a set X to set Y defined by ordered pairs $G = (x, y)$ such that $x \in X$ and $y \in Y$.
  - X is referred to as the *domain* of f, and Y is the *codomain*.
  - y is the *image* or *value* of f applied to the argument x.

Functions cont.

- Practically, a function is an operation that takes one or more inputs, and returns zero or more outputs.
  - For instance, the function $f(a, b)$ defined as $a + b$ takes two arguments, a and b, and returns a value $a + b$.
  - y can be the null set, in the sense that functions can return *nothing*.

Functions and Vectors

There are several ways to think about applying a function to a vector X_i of i values:

- Transformations
  - Element-wise Operations
  - Cumulative Operations
- Summaries
  - Point Summaries
  - Grouped Summaries

Transformation

The vector of all values in X_i, $[x_1, x_2, ...x_i]$ is in the domain of f, and a vector Y_i of equal length i is returned.

Examples/Use Cases per Datatype:

Dtype	Example
Numeric	Multiple Regression
Date/Time	Conversion to Timedeltas
Language	Translation

Transformation

$$f \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_i \end{bmatrix} \rightarrow \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_i \end{bmatrix}$$

Element-wise Operations

Element-wise operations are a special case of transformation. Individual elements of X_i are in the domain of f, and f is applied to each element of X_i to return a vector of length i where the ith element is the value of $f(x_i)$.

Examples/Use Cases per Datatype:

Dtype	Example
Numeric	Inversion (multiply by -1)
Date/Time	Timezone Conversion
Language	Stemming, capitalisation

Element-wise Operations: Illustration

$$f \odot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_i \end{bmatrix} \rightarrow \begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_i) \end{bmatrix}$$

Cumulative Operations

Cumulative operations apply a function with an expanding scope iteratively over a vector. Thus the first value is takes the first element as its argument, the second value takes the first two elements as arguments, the third value takes the first three elements as arguments, and so on.

Dtype	Example
Numeric	Cumulative Sum
Language	Grammar Parsing[1]

[1] One could construct an iterative parser that treats a language as a cumulative set of tokens; whether this parser would be the most effective is a different question.

Cumulative Operations: Illustration

$$f_{cum}(X_i) = \begin{bmatrix} f_1^1(x_1) \\ f_1^2(x_1, x_2) \\ \vdots \\ f_1^i(x_1, x_2, ..., x_i) \end{bmatrix} = \begin{bmatrix} f_1^1 x_i \\ f_1^2 x_i \\ \vdots \\ f_1^i x_i \end{bmatrix}$$

Summaries

A summary reduces a vector X_i of length i to a single value θ. Thus vector X_i is within the domain of f, and θ is value of f applied to X_i.

Examples/Use Cases per Datatype:

Dtype	Example(s)
Numeric	Mean, sum
Date/Time	Range, total seconds
Language	Sentiment scoring

Point Summaries

$$f \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_i \end{bmatrix} \rightarrow \theta$$

Grouped Summaries

Elements of $X_{i,g}$ are groupable in that each element x_i is a member of some group $g \in G$. In a sense they are somewhere between element-wise operations and summaries.

$$f \odot_g \begin{bmatrix} x_{1,g=1} \\ x_{2,g=1} \\ x_{3,g=2} \\ \vdots \\ x_{i,g} \end{bmatrix} \rightarrow \begin{bmatrix} f(x_{1,g=1}, x_{2,g=1}) \\ f(x_{3,g=2}) \\ \vdots \\ f(X_{i,g=g}) \end{bmatrix} \rightarrow \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_g \end{bmatrix}$$

Combining Data

Here are two different ways that two tabular datasets can be combined:

- Concatenating/Appending
- Joining/Merging

Concatenating

We can concatenate two or more tabular datasets so long as they all share at least one dimension ("height" or "width").

$$X = \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} \; Y = \begin{bmatrix} w & x \\ y & z \end{bmatrix}$$

$$XY = \begin{bmatrix} a & b \\ c & d \\ e & f \\ w & x \\ y & z \end{bmatrix}$$

Appending

The case of adding one dataset onto the “end” of another is referred to typically as “appending”.

Joining/Merging

Joining/merging is the combination of two datasets on one or more *keys*. In general, merging is done "horizontally". There are four kinds of joins:

- Inner
- Left
- Right
- Outer

Keys

Let's begin with the simple case of joining on a single key.

A key is a column, present in both X and Y, which usually shares some number of elements. These common elements are used to determine how the merge is conducted.

Our Data

Suppose we have two datasets. The first, X, details the first names and roles of a number of employees working on some project. The second, Y, details the names and employee ids of all full-time employees at that company.

$$X = \begin{bmatrix} \textbf{Role} & \textbf{Name} \\ \textit{Lead} & \textit{Bertha} \\ \textit{Research} & \textit{David} \\ \textit{Assistant} & \textit{Frankie} \\ \textit{Consultant} & \textit{Ryan} \end{bmatrix} \quad Y = \begin{bmatrix} \textbf{Name} & \textbf{id} \\ \textit{Adam} & 001 \\ \textit{Bertha} & 002 \\ \textit{Carla} & 003 \\ \textit{David} & 004 \\ \textit{Erica} & 005 \\ \textit{Frankie} & 006 \end{bmatrix}$$

Inner Join

An inner join retains rows that contain the *intersection* of the sets of the keys. In other words, it only retains rows from either dataframe that have a corresponding key in the other dataframe.

In the above example, the INNER JOIN of the two tables on NAME would result in the following table:

Role	**Name**	id
Lead	**Bertha**	002
Research	**David**	004
Assistant	**Frankie**	006

Left Join

A left join only retains rows that contain the keys of the left-hand dataset, inserting N/A where the right-hand key does not contain the corresponding left-hand one.

In the above example, the LEFT JOIN of the two tables on NAME would result in the following table. Note how Ryan's id is N/A.

Role	**Name**	id
Lead	**Bertha**	002
Research	**David**	004
Assistant	**Frankie**	006
Consultant	Ryan	N/A

Right Join

A right join retains rows that contain the keys of the right-hand dataset, inserting N/A where the left-hand key does not contain the corresponding right-hand one.

Role	Name	id
N/A	Adam	001
Lead	Bertha	002
N/A	Carla	003
Research	David	004
N/A	Erica	005
Assistant	Frankie	006

Outer Join

A (full) outer join retains all rows of both datasets, filling NAs where there is no key in common. NAs are inserted where a key does not exist in the opposing dataset.

Role	Name	id
N/A	Adam	001
Lead	**Bertha**	002
N/A	Carla	003
Research	**David**	004
N/A	Erica	005
Assistant	**Frankie**	006
Consultant	Ryan	N/A

Long- vs. Wide-Form Data

- "Long" and "wide" refer to two different ways of organising data that contains repeated observations of the same variable across (usually) time.
- In long format, each row represents an observation on a single unit at a single point in time. The temporal component is recorded in a separate column to the observations.
- In wide format, each row represents a single unit observed at multiple points in time. A separate column is used for each combination of time $\times$ quantity.

Example: Long Format

Constituency	Year	Party
Oxford East	2010	Labour
Oxford West	2010	Tory
Oxford East	2015	Labour
Oxford West	2015	Tory
Oxford East	2017	Labour
Oxford West	2017	LibDem
Oxford East	2019	Labour
Oxford West	2019	LibDem

Example: Wide Format

Constituency	Party2010	Party2015	Party2017	Party2019
Oxford East	Labour	Labour	Labour	Labour
Oxford West	Tory	Tory	LibDem	LibDem

Implementation

Functions in Python

Here's a simple function that adds 1 to the input:

```
def add_one(x):
    """
    This function adds 1 to the input.
    """
    y = x+1
    return y
```

`def add_one(x):`

```
def add_one(x):
    """
    This function adds 1 to the input.
    """
    y = x+1
    return y
```

- The command `def` followed by a space tells Python that you are defining a function.
- This function is given the name followed by `def`; in this case `add_one`.
- The *arguments* of the function are given after the function name, inside `()`.
- The `:` says that the definition line is done. The following line must be indented by four spaces.

Docstrings

- A string immediately after a function definition is automatically assigned as the **docstring** for that function.
- The docstring is the documentation that appears when you use the `func?` command.
- *This is optional*, but a great way to document your code. It also helps you remember and read your code faster.
- NB: I use a triple-double quote `"""` to create a multiline string. This is convenient, but not necessary (you can use a simple `"` or `'`).

Namespaces

- Python uses *namespaces* for variables.
- There are multiple levels of namespace, but the two relevant to you are *local* and *global*.
- Variables defined *within* a function are created within the *local* namespace of that function.
  - This means that they are only accessible from within the function.
- Variables defined *outside* a function are created within the *global* namespace.
- If a function contains a reference to a variable, it will first check to see whether the variable exists in the *local* namespace, and then the *global* one.

Local Variables not Accessibly Globally

The following code will result in an error:

```
def f(x):
    y = 5
    return x + y

print(y)
```

Local Accessed Before Global

The following code will return the *local* value of `y`, thus returning 10.

```
y = 0

def f(x):
    y = 5
    return x + y

print(f(5))
>> 10
```

Functions Reading from Global Variables

The following code uses `y`, which is defined globally. Therefore it returns 5.

```
y = 0

def f(x):
    return x + y

print(f(5))
>> 5
```

Lambda Functions

Python has *lambda functions*. These are essentially a way to define a function in-line. Below, the function `f` is equivalent to the line `lambda x: x+1`.

```
def f(x):
    return x+1


lambda x: x+1
```

Applying Functions to Pandas Objects

Remember that there are four relevant ways in which we might want to apply a function to a `Series` (or `DataFrame`!)

- Transformation
  - Element-wise Operation
  - Cumulative Operation
- Summary
  - Point Summary (covered in Lecture 2)
  - Grouped Summary

Element-wise Operations

The most general method for applying a function element-wise is the `apply` function.

- `apply` takes a function as its argument.
  - This can be a defined function, or a lambda function.
- When used with a `Series`, the function will be applied to each element of the vector.
- When used with a `DataFrame`, the function will be applied to each row (`axis=0`), or each column (`axis=1`) of the matrix.
  - In order to apply a function to every element of a `DataFrame`, use `applymap`.

`pd.Series.apply` with functions

The following two examples will add every element of a Series to itself. Note that the function is passed as an argument to `apply()` *without* `()`. This is because we are pointing to the function inside the `apply()`, not calling it.

```
def add_to_self(x):
    y = x + x
    return y

df['col1'].apply(add_to_self)
```

`pd.Series.apply` with lambda

I usually use `apply()` with *lambda functions*. There are two advantages to this approach:

- Conciseness: you do not have to define a new function for something you do only once.
- Passing arguments to the applied function: because `apply()` takes a function name, and not a call, as its argument, you cannot pass arguments to the function. By using a lambda function, this is possible:

pd.Series.apply with lambda examples

```
In [1]: import pandas as pd

In [2]: df = pd.DataFrame({'col1':range(5)})

In [4]: z=5

In [5]: df['col1'].apply(lambda x: x+x-z)
Out[5]:
0   -5
1   -3
2   -1
3    1
4    3

In [6]: def foo(x, z):
   ...:     y = x+x-z
   ...:     return y
```

Cumulative Operations

There are built-in methods for the four standard cumulative operations. These are methods of both `pd.Series` and `pd.DataFrame`.

- Sum: `cumsum`
- Product: `cumprod`
- Min: `cummin`
- Max: `cummax`
- *Advanced*: If you want to conduct other types of cumulative operation, you will need to use `pd.Series.expanding`, which works similarly to `groupby` (covered in subsequent slides).

`apply` versus for-loops

- You may have noticed that both kinds of transformations thus described could be done iteratively, with a for-loop.
- While this is true, the makers of `pandas` and its underlying library have optimised the library for *vectorized operations*.
- This essentially means that if you want to take advantage of the speed of `pandas`, you should use `apply` as much as possible when conducting transformations.
- This does not mean *never* use for-loops; the take-away should be that vectorization runs faster, but the fastest solution when accounting for programmer effort always depends on the task at hand.

Grouped Summaries

Pandas provides an extremely efficient and clean method for doing group summaries, but the syntax can be difficult to understand.

Imagine we have the following table:

Name	Location	Age	Female?	Likes BoJo (1-10)
Andy	Scotland	32	0	1
Barbara	Wales	48	1	3
Chris	Scotland	65	1	2
Dara	N. Ireland	55	0	6
Elaine	Wales	43	1	4

Groupby Syntax: Simple Group Operations

To conduct grouped summaries, we use the following syntax:

```
df.groupby('group_col')['value_col'].summary_func()
```

- `group_col` is the column we are grouping over.
- `value_col` is the column that contains the values we will be applying grouped summary functions to.
- `summary_func` is the function that that is applied to each group.

Groupby Syntax: Hierarchical Group Operations, Multiple Outputs

We can pass lists in place of the scalar values above:

```
df.groupby(['group_col1', 'group_col2'])[['value_col1', 'va
```

- If we pass a list of values to the `groupby()` function, the resulting groups are *hierarchical*, with the order of columns in the list passed to this function determining the rank of columns within the hierarchy.
- If we pass a list of values to the `[]` accessor following the `groupby()` call, we apply the grouped summary operation to each of the columns in this list. Order matters less here.

Groupby Example with Single Input

```
In [3]: df.groupby('Location')['Age'].mean()
Out[3]:
Location
N. Ireland    55.0
Scotland      48.5
Wales         45.5
Name: Age, dtype: float64
```

- This calculates the *mean age per location*.
  - `group_col`: Location
  - `value_col`: Age
  - `summary_func`: Mean
- Note the output is a `Series`, because the `value_cols` arguments contained only a single column.

Groupby Example with Hierarchical Groups, Single Input

```
In [4]: df.groupby(['Location', 'Female'])['PM_approval'].m
Out[4]:
Location    Female
N. Ireland  0          6.0
Scotland    0          1.0
            1          2.0
Wales       1          3.5
Name: PM_approval, dtype: float64
```

- This calculates the *mean PM approval per gender per location*.
  - "*Per gender per location*": For each area, get the per-gender mean.
  - In this case the hierarchical grouping goes Location over Gender.
- Note the nested index on the series; we discuss this later.

Groupby Example with Hierarchical Groups, Multiple Inputs

```
In [5]: df.groupby(['Location', 'Female'])[['Age', 'PM_appr
Out[5]:
                      Age  PM_approval
Location   Female
N. Ireland 0         55.0          6.0
Scotland   0         32.0          1.0
           1         65.0          2.0
Wales      1         45.5          3.5
```

- This calculates the *mean age and PM approval per gender per location*.
- Note that passing multiple input columns (i.e. a dataframe) returns a dataframe.

Append/Concat, Join/Merge

Pandas includes a multitude of functions for combining datasets, as well as an extensive guide with examples.

We cover just two functions:

- `pd.concat()`
- `pd.merge()`

Concatenation Syntax

To concatenate two or more dataframes, we use the following syntax:

```
pd.concat([df1, df2{, ..., dfn}], axis={0, 1})
```

- In the above, `{}` indicates that the argument is optional.
- When `axis=0`, dataframes are stacked "vertically". Where they have columns in common, the columns will be "stacked", otherwise `N/A` will be inserted in the cells.
  - If the columns are in a different order, make sure to pass `sort=True`.
- When `axis=1`, dataframes are stacked "horizontally". Where the index aligns, rows will be concatenated side-by-side. If there are index values not common between the dataframes, then `N/A` will be inserted in the cells.

Merge Syntax

```
pd.merge(
    left_df,
    right_df,
    how={'left', 'right', 'outer', 'inner'},
    {on=common_key},
    {left_on=left_key},
    {right_on=right_key},
    {left_index={True, False}},
    {right_index={True, False}}
)
```

- The first three arguments are straightforward:
  - `left_df` is the left-hand dataframe to be merged.
  - `right_df` is the right-hand dataframe to be merged.
  - `how` determines what kind of join is used. (See previous slides.)

Merge Syntax cont.

- The following argument(s) identify the keys from the left- and right-hand dataframes to be used for the join.
  - `on` can be used when the key is a column with the same name in both dataframes.
  - `left_on` should be used in conjunction with `right_on` or `right_index`. Each can be a single column name or a list of column names to be used as merge keys.
  - `left_index` and `right_index` take either `True` or `False`. If `True`, then the index of the according dataframe is used as the merge key.

Performance

- In general, indices are faster to perform operations on, but also require more memory to store.
- Pre-sorting concatenation axes or merging keys will improve performance greatly.

Long and Wide Format

Pandas provides two functions for changing between long and wide data format. For further details I direct you to the documentation:

- `pd.pivot()`: Converts long to wide.
- `pd.melt()`: Converts wide to long.

Coding Tutorial

Let's get to some advanced pandas!