Pandas

Introduction

`pandas` is the Python Data Analysis package. It allows for data ingestion, transformation and cleaning, and creates objects that can then be passed on to analytic packages like `statsmodels` and `scikit-learn` for modeling and packages like `matplotlib`, `seaborn`, and `plotly` for visualization.

`pandas` is built on top of numpy, so many numpy functions are commonly used in manipulating `pandas` objects.

> *`pandas` is a pretty extensive package, and we'll only be able to cover some of its features. For more details, there is free online documentation at pandas.pydata.org. You can also look at the book "Python for Data Analysis (2nd edition)" by Wes McKinney, the original developer of the pandas package, for more details.*

Starting pandas

As with any Python module, you have to "activate" `pandas` by using `import`. The "standard" alias for `pandas` is `pd`. We will also import `numpy`, since `pandas` uses some `numpy` functions in the workflows.

```
import numpy as np
import pandas as pd
```

Data import and export

Most data sets you will work with are set up in tables, so are rectangular in shape. Think Excel spreadsheets. In `pandas` the structure that will hold this kind of data is a `DataFrame`. We can read external data into a `DataFrame` using one of many `read_*` functions. We can also write from a `DataFrame` to a variety of formats using `to_*` functions. The most common of these are listed below:

Format type	Description	reader	writer
text	CSV	read_csv	to_csv
	Excel	read_excel	to_excel
text	JSON	read_json	to_json
binary	Feather	read_feather	to_feather
binary	SAS	read_sas	
SQL	SQL	read_sql	to_sql

We'll start by reading in the `mtcars` dataset stored as a CSV file

```
pd.read_csv('data/mtcars.csv')
```

This just prints out the data, but then it's lost. To use this data, we have to give it a name, so it's stored in Python's memory

```
mtcars = pd.read_csv('data/mtcars.csv')
```

> *One of the big differences between a spreadsheet program and a programming language from the data science perspective is that you have to load data into the programming language. It's not "just there" like Excel. This is a good thing, since it allows the common functionality of the programming language to work across multiple data sets, and also keeps the original data set pristine. Excel users can run into problems and corrupt their data if they are not careful.*

If we wanted to write this data set back out into an Excel file, say, we could do

```
mtcars.to_excel('data/mtcars.xlsx')
```

> *You may get an error if you don't have the `openpyxl` package installed. You can easily install it from the Anaconda prompt using `conda insall openpyxl` and following the prompts.*

Exploring a data set

We would like to get some idea about this data set. There are a bunch of functions linked to the `DataFrame` object that help us in this. First we will use `head` to see the first 8 rows of this data set

```
mtcars.head(8)
```

This is our first look into this data. We notice a few things. Each column has a name, and each row has an *index*, starting at 0.

> *If you're interested in the last N rows, there is a corresponding `tail` function*

Let's look at the data types of each of the columns

```
mtcars.dtypes
```

This tells us that some of the variables, like `mpg` and `disp`, are floating point (decimal) numbers, several are integers, and `make` is an "object". The `dtypes` function borrows from `numpy`, where there isn't really a type for character or categorical variables. So most often, when you see "object" in the output of `dtypes`, you think it's a character or categorical variable.

We can also look at the data structure in a bit more detail.

```
mtcars.info()
```

This tells us that this is indeed a `DataFrame`, wth 12 columns, each with 32 valid observations. Each row has an index value ranging from 0 to 11. We also get the approximate size of this object in memory.

You can also quickly find the number of rows and columns of a data set by using `shape` which is borrowed from numpy.

```
mtcars.shape
```

More generally, we can get a summary of each variable using the `describe` function

```
mtcars.describe()
```

These are usually the first steps in exploring the data.

Data structures and types

pandas has two main data types: `Series` and `DataFrame`. These are analogous to vectors and matrices, in that a `Series` is 1-dimensional while a `DataFrame` is 2-dimensional.

pd.Series

The `Series` object holds data from a single input variable, and is required, much like numpy arrays, to be homogeneous in type. You can create `Series` objects from lists or numpy arrays quite easily

```
s = pd.Series([1,3,5,np.nan, 9, 13])
s
```

```
s2 = pd.Series(np.arange(1,20))
s2
```

You can access elements of a `Series` much like a `dict`

```
s2[4]
```

There is no requirement that the index of a `Series` has to be numeric. It can be any kind of scalar object

```
s3 = pd.Series(np.random.normal(0,1, (5,)), index = ['a','b','c','d','e'])
s3
```

```
s3['d']
```

You can extract the actual values into a numpy array

```
s3.to_numpy()
```

pd.DataFrame

The `DataFrame` object holds a rectangular data set. Each column of a `DataFrame` is a `Series` object. This means that each column of a `DataFrame` must be comprised of data of the same type, but different columns can hold data of different types. This structure is extremely useful in practical data science. The invention of this structure was, in my opinion, transformative in making Python an effective data science tool.

The `DataFrame` can be created by importing data, as we saw in the previous section. It can also be created by a few methods within Python.

First, it can be created from a 2-dimensional `numpy` array.

```
rng = np.random.RandomState(25)
d1 = pd.DataFrame(rng.normal(0,1, (4,5)))
d1
```

You will notice that it creates default column names, that are merely the column number, starting from 0. We can actually change the column names (which can be extracted and replaced with the `columns` attribute.

```
d1.columns
```

```
d1.columns = pd.Index(['V'+str(i) for i in range(1,6)]) # Index creates the right objects for both column names and row names, which can be extracted and changed with the `index` attribute
d1
```

Exercise: Can you explain what I did in the list comprehension above? The key points are understanding `str` and how I constructed the `range`.

You can also extract data from a homogeneous `DataFrame` to a `numpy` array

```
d1.to_numpy()
```

The other easy way to create a `DataFrame` is from a `dict` object, where each component object is either a list or a numpy array, and is homogeneous in type. One exception is if a component is of size 1; then it is repeated to meet the needs of the `DataFrame`'s dimensions

```
df = pd.DataFrame({
    'A':3.,
    'B':rng.random_sample(5),
    'C': pd.Timestamp('20200512'),
    'D': np.array([6] * 5),
    'E': pd.Categorical(['yes','no','no','yes','no']),
    'F': 'NIH'})
df
```

```
df.info()
```

We note that C is a date object, E is a category object, and F is a text/string object. pandas has excellent time series capabilities (having origins in FinTech), and the `TimeStamp` function creates datetime objects which can be queried and manipulated in Python. We'll describe category data in the next section.

You can extract particular columns of a `DataFrame` by name

```
df['E']
```

```
df['B']
```

```

```

Categorical data

`pandas` provides a `Categorical` function and a `category` object type to Python. This type is analogous to the `factor` data type in R. It is meant to address categorical or discrete variables, where we need to use them in analyses. Categorical variables typically take on a small number of unique values, like gender, blood type, country of origin, race, etc.

You can create categorical `Series` in a couple of ways:

```
s = pd.Series(['a','b','c'], dtype='category')
```

```
df['F'].astype('category')
```

You can also create `DataFrame`'s where each column is categorical

```
df = pd.DataFrame({'A': list('abcd'), 'B': list('bdca')})
df_cat = df.astype('category')
df_cat.dtypes
```

You can explore categorical data in a variety of ways

```
df_cat['A'].describe()
```

```
df['A'].value_counts()
```

One issue with categories is that, if a particular level of a category is not seen before, it can create an error. So you can pre-specify the categories you expect

```
df_cat['B'] = pd.Categorical(list('aabb'), categories = ['a','b','c','d'])
df_cat['B'].value_counts()
```

Missing data

Both `numpy` and `pandas` allow for missing values, which are a reality in data science. The missing values are coded as `np.nan`. Let's create some data and force some missing values

```
df = pd.DataFrame(np.random.randn(5, 3), index = ['a','c','e', 'f','g'], columns = ['one','two','three']) # pre-specify index and column names
df['four'] = 20 # add a column named "four", which will all be 20
df['five'] = df['one'] > 0
df
```

```
df2 = df.reindex(['a','b','c','d','e','f','g'])
df2
```

The code above is creating new blank rows based on the new index values, some of which are present in the existing data and some of which are missing

```
df2.isna()
```

```
df2['one'].notna()
```

Getting complete data

```
df2.dropna(how='any')
```

Filling missing values

```
df2.fillna(value = 5)
```

Fill with the column mean

```
df3 = df2.copy()
df3 = df3.select_dtypes(exclude=[object])   # remove non-numeric columns
df3.fillna(df3.mean())  # df3.mean() computes column-wise means
```

Data transformation

Arithmetic operations

If you have a `Series` or `DataFrame` that is all numeric, you can add or multiply single numbers to all the elements together.

```
A = pd.DataFrame(np.random.randn(4,5))
print(A)
```

```
print(A + 6)
```

```
print(A * -10)
```

If you have two compatible (same dimension) numeric `DataFrame`s, you can add, subtract, multiply and divide elementwise

```
B = pd.DataFrame(np.random.randn(4,5) + 4)
print(A + B)
```

```
print(A * B)
```

If you have a `Series` with the same number of elements as the number of columns of a `DataFrame`, you can do arithmetic operations, with each element of the `Series` acting upon each column of the `DataFrame`

```
c = pd.Series([1,2,3,4,5])
print(A + c)
```

```
print(A * c)
```

Extracting rows and columns

There are two extractor functions in `pandas`:

- `loc` extracts by label (index label, column label, slice of labels, etc.
- `iloc` extracts by index (integers, slice objects, etc.

```
df2.loc['a',:]
```

```
df2.loc[:, 'three'] # or df['three']
```

```
df2.loc['a':'c']
```

> *Note, in pandas, this slice selector includes the smallest **and largest** index, unlike in other Python and `numpy` uses, where the largest index is omitted*

```
df2.iloc[0:2, :]
```

To add to the confusion, the normal slicing rules apply when we use indices rather than labels.

```
df2.iloc[0:2, 1:4]
```

```
df2.loc['a':'b', 'two':'four']
```

As a convenience, if you just use a slice by numbers in a `DataFrame`, it will slice the rows

```
df2[:3]
```

In contrast, if you supply a single label, it will extract the column!!

```
df2['two']
```

Boolean selection

```
df2[df2['one'] > 0]
```

```
df2[(df2['one'] > 0) & (df2['three'] < 0)]
```

`query`

`DataFrame`'s have a `query` method allowing selection using a Python expression

```
n = 10
df = pd.DataFrame(np.random.rand(n, 3), columns = list('abc'))
df
```

```
df[(df['a'] < df['b']) & (df['b'] < df['c'])]
```

```
df.query('(a < b) & (b < c)')
```

Concatenation of data sets

Let's create some example data sets

```
df1 = pd.DataFrame({'A': ['a'+str(i) for i in range(4)],
    'B': ['b'+str(i) for i in range(4)],
    'C': ['c'+str(i) for i in range(4)],
    'D': ['d'+str(i) for i in range(4)]})

df2 =  pd.DataFrame({'A': ['a'+str(i) for i in range(4,8)],
    'B': ['b'+str(i) for i in range(4,8)],
    'C': ['c'+str(i) for i in range(4,8)],
    'D': ['d'+str(i) for i in range(4,8)]})
df3 =  pd.DataFrame({'A': ['a'+str(i) for i in range(8,12)],
    'B': ['b'+str(i) for i in range(8,12)],
    'C': ['c'+str(i) for i in range(8,12)],
    'D': ['d'+str(i) for i in range(8,12)]})
```

We can concatenate these `DataFrame` objects by row

```
row_concatenate = pd.concat([df1, df2, df3])
print(row_concatenate)
```

This stacks the dataframes together. They are literally stacked, as is evidenced by the index values being repeated.

```
row_concatenate.iloc[3,:]
```

Exercise: What happens if you replace `iloc` with `loc`?

This same exercise can be done by the `append` function

```
df1.append(df2).append(df3)
```

Suppose we want to append a new row to `df1`. Lets create a new row.

```
new_row = pd.Series(['n1','n2','n3','n4'])
pd.concat([df1, new_row])
```

That's a lot of missing values. The issue is that the we don't have column names in the `new_row`, and the indices are the same, so pandas tries to append it my making a new column. The solution is to make it a `DataFrame`.

```
new_row = pd.DataFrame([['n1','n2','n3','n4']], columns = ['A','B','C','D'])
print(new_row)
```

```
pd.concat([df1, new_row])
```

or

```
df1.append(new_row)
```

Adding columns

```
pd.concat([df1,df2,df3], axis = 1)
```

The option `axis=1` ensures that concatenation happens by columns. The default value `axis = 0` concatenates by rows.

Let's play a little game. Let's change the column names of `df2` and `df3` so they are not the same as `df1`.

```
df2.columns = ['E','F','G','H']
df3.columns = ['A','D','F','H']
pd.concat([df1,df2,df3])
```

Now pandas ensures that all column names are represented in the new data frame, but with missing values where the row indices and column indices are mismatched. Some of this can be avoided by only joining on common columns. This is done using the `join` option ir `concat`. The default value is 'outer`, which is what you see. above

```
pd.concat([df1, df3], join = 'inner')
```

You can do the same thing when joining by rows, using `axis = 1` and `join="inner"` to only join on rows with matching indices. Reminder that the indices are just labels and happen to be the row numbers by default.

Merging data sets

For this section we'll use a set of data from a survey, also used by Daniel Chen in "Pandas for Everyone"

```
person = pd.read_csv('data/survey_person.csv')
site = pd.read_csv('data/survey_site.csv')
survey = pd.read_csv('data/survey_survey.csv')
visited = pd.read_csv('data/survey_visited.csv')
```

```
print(person)
```

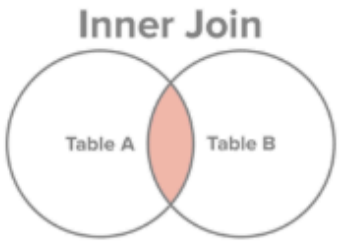
```
print(site)
```

```
print(survey)
```

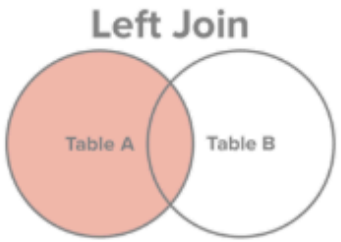
```
print(visited)
```

There are basically four kinds of joins:

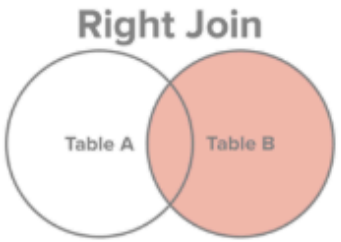
pandas	R	SQL	Description
left	left_join	left outer	keep all rows on left
right	right_join	right outer	keep all rows on right
outer	outer_join	full outer	keep all rows from both
inner	inner_join	inner	keep only rows with common keys



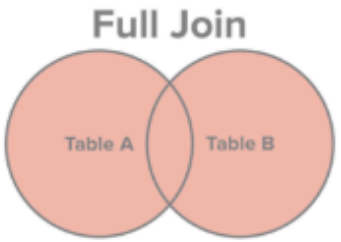
Select all records from Table A and Table B, where the join condition is met.



Select all records from Table A, along with records from Table B for which the join condition is met (if at all).



Select all records from Table B, along with records from Table A for which the join condition is met (if at all).



Select all records from Table A and Table B, regardless of whether the join condition is met or not.

The terms `left` and `right` refer to which data set you call first and second respectively.

We start with an left join

```
s2v_merge = survey.merge(visited, left_on = 'taken',right_on = 'ident', how = 'left')
```

```
print(s2v_merge)
```

Here, the left dataset is `survey` and the right one is `visited`. Since we're doing a left join, we keed all the rows from `survey` and add columns from `visited`, matching on the common key, called "taken" in one dataset and "ident" in the other. Note that the rows of `visited` are repeated as needed to line up with all the rows with common "taken" values.

We can now add location information, where the common key is the site code

```
s2v2loc_merge = s2v_merge.merge(site, how = 'left', left_on = 'site', right_on = 'name')
print(s2v2loc_merge)
```

Lastly, we add the person information to this dataset.

```
merged = s2v2loc_merge.merge(person, how = 'left', left_on = 'person', right_on = 'ident')
print(merged.head())
```

You can merge based on multiple columns as long as they match up.

```
ps = person.merge(survey, left_on = 'ident', right_on = 'person')
vs = visited.merge(survey, left_on = 'ident', right_on = 'taken')
print(ps)
```

```
print(vs)
```

```
ps_vs = ps.merge(vs,
                 left_on = ['ident','taken', 'quant','reading'],
                 right_on = ['person','ident','quant','reading']) # The keys need to correspond
ps_vs.head()
```

Note that since there are common column names, the merge appends `_x` and `_y` to denote which column came from the left and right, respectively.

Tidy data principles and reshaping datasets

The tidy data principle is a principle espoused by Dr. Hadley Wickham, one of the foremost R developers. Tidy data is a structure for datasets to make them more easily analyzed on computers. The basic principles are

- Each row is an observation
- Each column is a variable
- Each type of observational unit forms a table

Melting (unpivoting) data

Melting is the operation of collapsing multiple columns into 2 columns, where one column is formed by the old column names, and the other by the corresponding values. Some columns may be kept fixed and their data are repeated to maintain the interrelationships between the variables.

We'll start with loading some data on income and religion in the US from the Pew Research Center.

```
pew = pd.read_csv('data/pew.csv')
print(pew.head())
```

This dataset is considered in "wide" format. There are several issues with it, including the fact that column headers have data. Those column headers are income groups, that should be a column by tidy principles. Our job is to turn this dataset into "long" format with a column for income group.

We will use the function `melt` to achieve this. This takes a few parameters:

- **id_vars** is a list of variables that will remain as is
- **value_vars** is a list of column nmaes that we will melt (or unpivot). By default, it will melt all columns not mentioned in id_vars
- **var_name** is a string giving the name of the new column created by the headers (default: `variable`)
- **value_name** is a string giving the name of the new column created by the values (default: `value`)

```
pew_long = pew.melt(id_vars = ['religion'], var_name = 'income_group', value_name = 'count')
print(pew_long.head())
```

Separating columns containing multiple variables

We will use an Ebola dataset to illustrate this principle

```
ebola = pd.read_csv('data/country_timeseries.csv')
print(ebola.head())
```

Note that for each country we have two columns – one for cases (number infected) and one for deaths. Ideally we want one column for country, one for cases and one for deaths.

The first step will be to melt this data sets so that the column headers in question from a column and the corresponding data forms a second column.

```
ebola_long = ebola.melt(id_vars = ['Date','Day'])
print(ebola_long.head())
```

We now need to split the data in the `variable` column to make two columns. One will contain the country name and the other either Cases or Deaths. We will use some string manipulation functions that we will see later to achieve this.

```
variable_split = ebola_long['variable'].str.split('_', expand=True) # split on the `_` character
print(variable_split[:5])
```

The `expand=True` option forces the creation of an `DataFrame` rather than a list

```
type(variable_split)
```

We can now concatenate this to the original data

```
variable_split.columns = ['status','country']

ebola_parsed = pd.concat([ebola_long, variable_split], axis = 1)

ebola_parsed.drop('variable', axis = 1, inplace=True) # Remove the column named "variable" and replace the old data with the new one in the same location

print(ebola_parsed.head())
```

Pivot/spread datasets

If we wanted to, we could also make two columns based on cases and deaths, so for each country and date you could easily read off the cases and deaths. This is achieved using the `pivot_table` function.

```
ebola_parsed.pivot_table(index = ['Date','Day', 'country'], columns = 'status', values = 'value')
```

This creates something called `MultiIndex` in the `pandas` `DataFrame`. This is useful in some advanced cases, but here, we just want a normal `DataFrame` back. We can achieve that by using the `reset_index` function.

```
ebola_parsed.pivot_table(index = ['Date','Day','country'], columns = 'status', values = 'value').reset_index()
```

In the `pivot_table` syntax, `index` refers to the columns we don't want to change, `columns` refers to the column whose values will form the column names of the new columns, and `values` is the name of the column that will form the values in the pivoted dataset.

Pivoting is a 2-column to many-column operation, with the number of columns formed depending on the number of unique values present in the column of the original data that is entered into the `columns` argument of `pivot_table`

Exercise: Load the file `weather.csv` into Python and work on making it a tidy dataset. It requires melting and pivoting. The dataset comprises of the maximun and minimum temperatures recorded each day in 2010. There are lots of missing value. Ultimately we want columns for days of the month, maximum temperature and minimum tempearture along with the location ID, the year and the month.

Data aggregation and split-apply-combine

Aggregation

We'll use the Gapminder dataset for this section

```
df = pd.read_csv('data/gapminder.tsv', sep = '\t') # data is tab-separated, so we use `\t` to specify that

```