Crash Course in Python; Unit 2

Data Management with Numpy and Pandas

UNIVERSITY OF CALGARY

Learning Outcomes

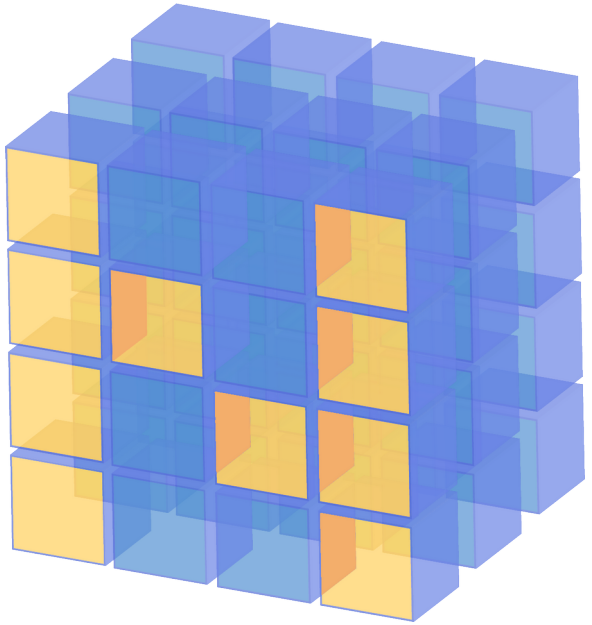
By the end of the module, you will be able to:

- Run common mathematical operators using Numpy
- Load spreadsheet data into Python using Panda's dataframes
- Summarize the loaded data, using Numpy with Pandas
- Query the data by both position and labels
- Add to, delete, re-organize, and modify said data
- Save the now modified data to a file for later use

All Python code described and used in this slideshow will be provided in a Jupyter Notebook file, which you can run on TALC to further experiment with. A (non-runnable) PDF version of said code will also be available for reference.

Numpy and Pandas

- Numpy is a numerical library for Python, designed to allow for more complex and efficient mathematical analyses on large datasets
- Pandas is another numerical library which builds off of Numpy, focusing on large scale data management through data structures known as a DataFrame
- Both are actively developed, and nearly ubiquitous in modern data science applications



pandas

https://user-images.githubusercontent.com/98330/63813335-20cd4b80-c8e2-11e9-9c04-e4dbf7285aa1.png, https://pandas.pydata.org/static/img/pandas_secondary.svg

Loading External Python Packages

- To load a package, use the '`import`' statement followed by the package you wish to load:

```
~ import numpy
~ import pandas
```

- With Numpy and Pandas in particular, it is common to alias them as '`np`' and '`pd`', respectively. This can be done using the '`as`' statement in Python, following the original import:

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

- If only part of a package is needed, the '`from`' statement can be used to select that element on its own:

```
~ from numpy import floor
```

Using Loaded Packages

- Once loaded, the components of the package can be used as though they were functions of that package's name:

```
~ import numpy as np
~ tmp_data = [1, 2, 3, 4]
~ # 'sum', as the name implies, takes the sum
~ tmp_sum = np.sum(tmp_data)
~ print(tmp_sum)
> 10
~ # 'cumprod' is short for 'cummulative product'
~ print(np.cumprod(tmp_data))
> [1 2 6 24]
```

- A full list of all functions for Numpy is available at https://numpy.org/doc/stable/reference/routines.html

Importing Data with Pandas

- Once pandas is imported, Excel spreadsheets can be imported using its '`read_excel`' function:

```
~ df = pd.read_excel('demo_data.xlsm')
```

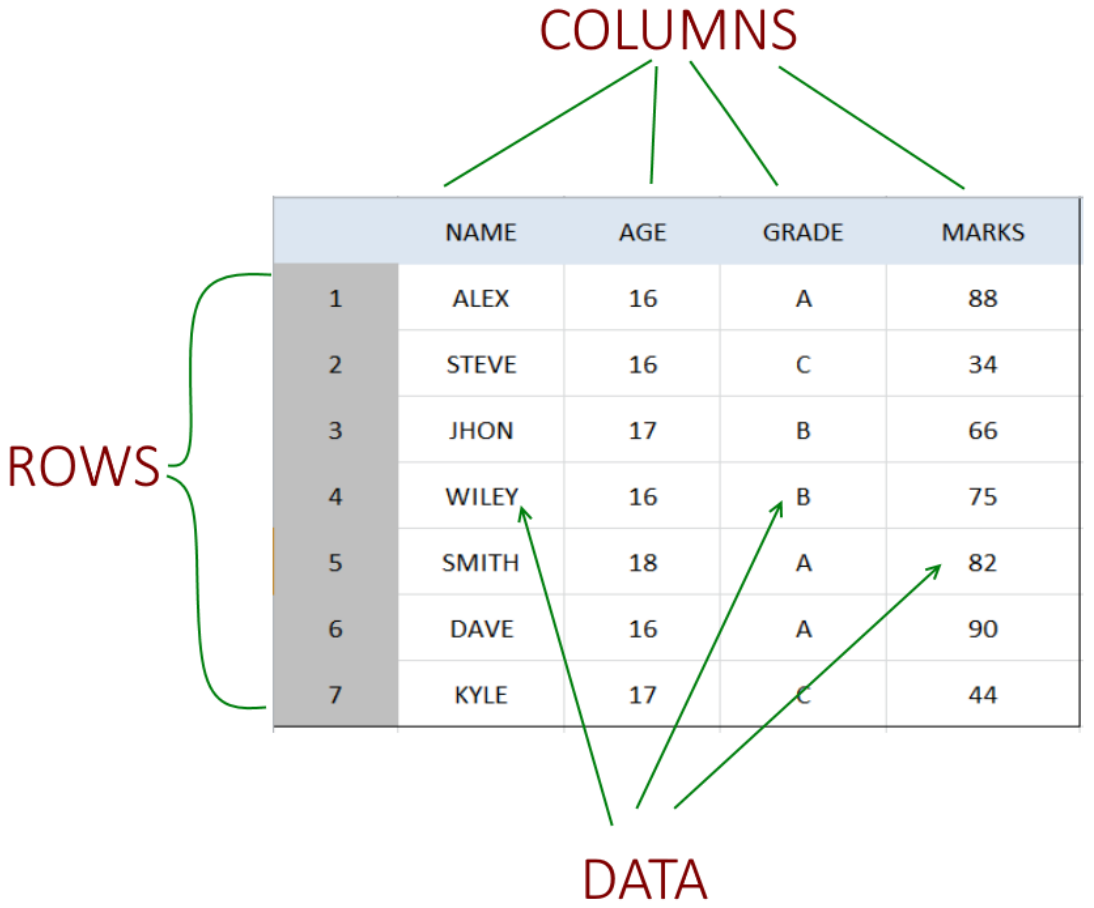
- If the file is in csv (comma-separated value) format, '`read_csv`' is used instead. Importing tsv (tab-separated value) format is similar, but requires the addition of the '`sep="\t"`' argument:

```
~ # csv file
~ df = pd.read_csv('demo_data.csv')
~ # tsv file
~ df2 = pd.read_csv('demo_data.tsv', sep='\t')
```

- The imported data is stored within a DataFrame structure, often shortened to '`df`', for later use within the program

DataFrame Structure

- DataFrames are analogous to tables within a spreadsheet, but much more efficient and powerful
- Each DataFrame (shortened to ‘`df`’ here) have three primary elements:
  - The row indices, which label the rows of the table (accessed with ‘`df.index`’)
  - The column headers, which label each column in the data (accessed with ‘`df.columns`’)
  - The data itself, which can be queried use the prior two



	NAME	AGE	GRADE	MARKS
1	ALEX	16	A	88
2	STEVE	16	C	34
3	JHON	17	B	66
4	WILEY	16	B	75
5	SMITH	18	A	82
6	DAVE	16	A	90
7	KYLE	17	C	44

UNIVERSITY OF CALGARY

https://1.bp.blogspot.com/-iDCBVemkdJQ/XvwkfTCHg3I/AAAAAAAAELI/oLWwAYFl1iAvu42iE9nQdNgYHaJX7CV6gCK4BGAsYHg/s991/PWANDAS%2BDATAFRAME.png

DataFrame Querying

UNIVERSITY OF CALGARY

Viewing DataFrame Contents

- Like any other object, DataFrames can be viewed (often in a truncated manner) using the '`print`' statement:

```
~ df = pd.read_excel('demo_data.xlsm')
~ print(df)
>    ID Allele Sex (Biological)  Age
> 0   1      a                 m   21
> 1   2      a                 m   74
> 2   3      b                 m   63
> 3   4      a                 f   35
> 4   5      b                 f   74
> 5   6      b                 m   73
> 6   7      b                 f   50
> 7   8      a                 m   66
> 8   9      b                 f   35
> 9  10      a                 f   36
```

Querying DataFrames by Position (list-like slicing)

- DataFrames are queried similarly to lists using '`iloc`', taking two numbers/slices in `[row, column]` form. Note that queries that result in more than one row or column produce a new DataFrame

```
~ # Single element
~ df.iloc[0, 2]
> 'm'
~ # All on the 2nd row
~ df.iloc[1, :]
> ID                    2
>              ...
> Age                  74
~ # All element's on rows 2-4, columns 2 and 3
~ df.iloc[1:4, [1, 2]]
>   Allele Sex (Biological)
> 1      a                m
> 2      b                m
> 3      a                f
```

Querying DataFrames by Label (Database-like)

- DataFrames can be queried by their column and row index values (rather than position) using '`loc`'. The query structure is otherwise the same:

```
~ # Single element
~ df.loc[0, 'Age']
> 21
~ # All elements for column 'Allele'
~ df.loc[:, "Allele"]
> 0    a
>    ...
> 9    a
~ # The age and sex (in that order!) for row indices 6-8
~ df.loc[6:8, ["Age", "Sex (Biological)"]]
>    Age Sex (Biological)
> 6   50                f
> 7   66                m
> 8   35                f
```

Querying DataFrames by Condition (Filtering)

- DataFrames can be filtered by a condition, similar to an '`if`' statement. To do so, use dataframe alongside the desired condition as the index for the query (no '`loc`' or '`iloc`'):

```
~ # Select only records for allele 'b'
~ print(df[df.loc[:, "Allele"] == "b"])
>    ID Allele Sex (Biological)  Age
> 2   3      b                 m   63
>                   ...
> 8   9      b                 f   35
~ # Select only records for males over 50 years old
~ # Note we use '&' instead of 'and', plus additional brackets
~ print(df[(df.loc[:, "Age"] > 50) & (df.loc[:, "Sex (Biological)"]
== "m"])])
>    ID Allele Sex (Biological)  Age
> 1   2      a                 m   74
>                   ...
> 7   8      a                 m   66
```

Summarizing the Entire DataFrame

- All *numerical* data within the dataframe can be summarized using the '`describe`' function. Note that this description is a DataFrame as well:

```
~ print(df.describe())
>                ID        Age
> count   10.00000   10.00000
> mean     5.50000   52.70000
> std      3.02765   19.77681
> min      1.00000   21.00000
> 25%      3.25000   35.25000
> 50%      5.50000   56.50000
> 75%      7.75000   71.25000
> max     10.00000   74.00000
```

Manipulating DataFrame Data

UNIVERSITY OF CALGARY

Removing Rows/Columns with ‘drop’

- The ‘`drop`’ statement requires three arguments to function:
  - ‘*`axis`*’: Set to 0 if you want to drop row(s), 1 if you want to drop column(s)
  - ‘*`labels`*’: Specifies the rows/columns to drop. Can also be single row/column, or a list of rows/columns. These must be the labels of the rows/columns, not their positional location
  - ‘*`inplace`*’: If true, the DataFrame will change itself permanently, while if false the DataFrame will just return a copy of itself with the changes applied (which you can save to a variable if desired)

https://icons555.com/images/icons-red/image_icon_delete_database_pic_512x512.png

‘drop’ in Action

- ‘drop’ without ‘inplace’, single column:

```
~ df2 = df.drop('Age', axis=1)
~ print(df2)
>    ID Allele Sex (Biological)
> 0   1      a                 m
>             ...
> 9  10      a                 f
```

- ‘drop’ with ‘inplace’, alternating rows, using the DataFrame generated prior:

```
~ df2.drop([1,3,5,7,9], axis=0, inplace=True)
~ print(df2)
>    ID Allele Sex (Biological)
> 0   1      a                 m
> 2   3      b                 m
> 4   5      b                 f
> 6   7      b                 f
> 8   9      b                 f
```

Inserting New Data

- Like variables, we can assign new data to the DataFrame via the assignment operator '='. We simply need to specify the context of the insertion, similar to how we would query the dataframe:

```
~ # Insert a new column with the header "Name"
~ # NOTE: The size of the inserted list must be the same as
that of the DataFrame along the respective axis!
~ df2.loc[:, "Name"] = ["Alex", "Barnie", "Charles", "Dixie",
"Ethan"]
~ print(df2)
>    ID Allele Sex (Biological)     Name
> 0   1      a                  m     Alex
> 2   3      b                  m   Barnie
> 4   5      b                  f  Charles
> 6   7      b                  f    Dixie
> 8   9      b                  f    Ethan
```

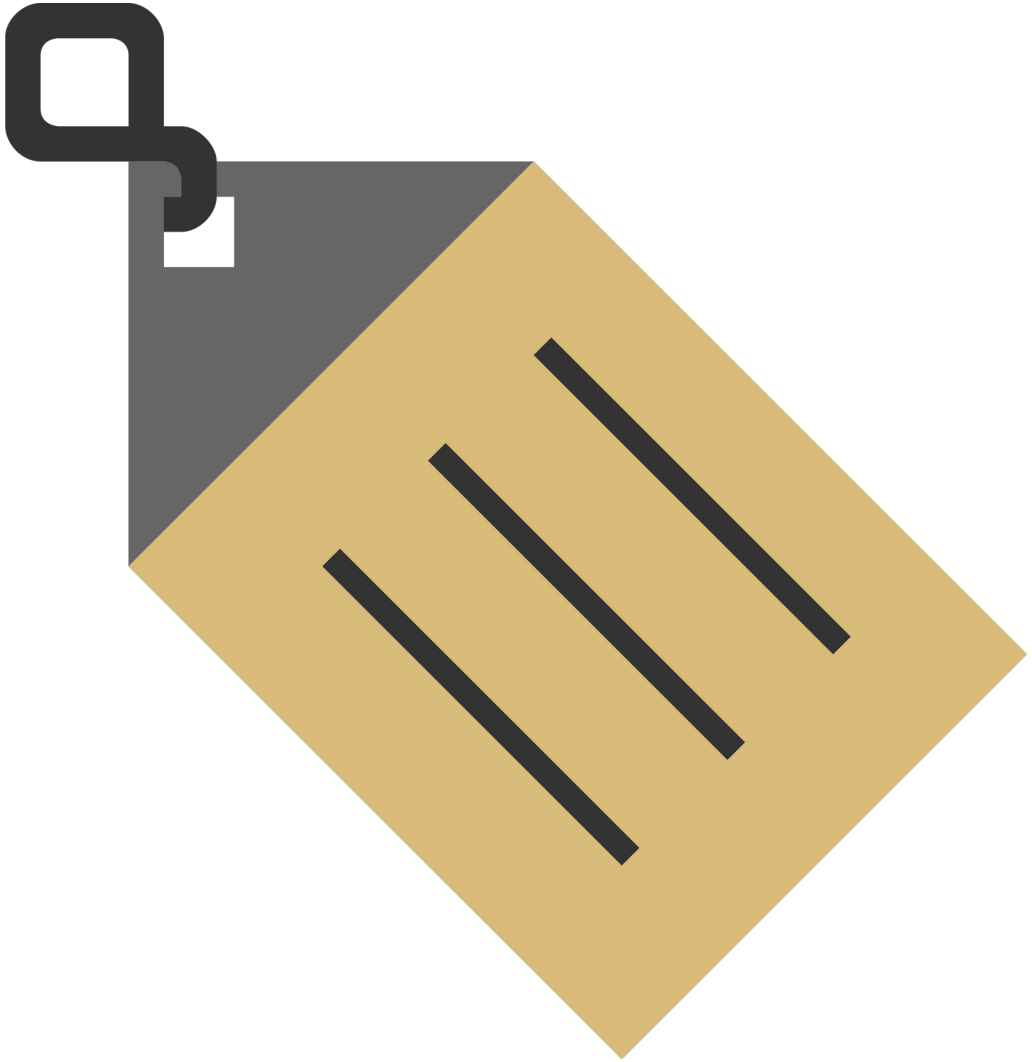
Replacing DataFrame Data

- Existing contents in a DataFrame can be replaced with identical syntax to an insertion; simply specify the existing element(s) that should be overridden

```
~ # Replace "Ethan" with "Earl"
~ df2.loc[8, "Name"] = "Earl"
~ print(df2)
>    ID Allele Sex (Biological)     Name
> 0   1      a                m     Alex
> 2   3      b                m   Barnie
> 4   5      b                f  Charles
> 6   7      b                f    Dixie
> 8   9      b                f     Earl
```

Changing Row Labels with ‘set_index’

- We can request that a DataFrame use a column its row indices by using the ‘`set_index`’ function. This requires three arguments to work:
  - ‘*`keys`*’: The new column to set as an index. Note that a list of desired columns can be used to have their combinations treated as the index, though this complicates later querying
  - ‘*`drop`*’: Specifies whether the original column should be deleted in the DataFrame when it becomes the index.
  - ‘*`inplace`*’: Whether the original DataFrame should modify itself permanently, or just return a copy of itself with the now-changed row indices



https://clipground.com/images/name-tag-clipart-6.jpg

‘set_index’ in Action

- Index set without ‘inplace’ or ‘drop’, single column:

```
~ df3 = df2.set_index(‘ID’, drop=False, inplace=False)
~ print(df3)
>      ID Allele Sex (Biological)      Name
> ID
> 1     1       a                 m      Alex
>                        ...
> 9     9       b                 f      Earl
```

- Index set with ‘inplace’ and ‘drop’, multiple columns (forming a multi-index):

```
~ df3.set_index([‘Name’, ‘Allele’], drop=True, inplace=True)
~ print(df3)
>                    ID Sex (Biological)
> Name    Allele
> Alex    a         1                 m
>                  ...
> Earl    b         9                 f
```

Setting Row and Column Indices Directly

- We can also provide a completely new set of indices for the rows/columns of a DataFrame by assigning a list of elements to the DataFrame's '`index`' and '`columns`' parameter, respectively:

```
~ # Update the row's labels to represent a study ID
~ # NOTE; much like data insertion, the size of the list must
match the number of rows (index) or columns (columns)!
~ df3.index = [1242, 1289, 1308, 1415, 1417]
~ print(df3)
>      Sex (Biological)     Name
> 1242                m     Alex
>                 ...
> 1417                f     Earl
~ # The now explicitly set index can be used for queries
~ print(df3.loc[1308, "Name"])
> Charles
```

Transposing (Flipping) the DataFrame

- Sometimes we need to swap our rows and columns to better organize the data, or to prepare it for some analysis. This is called *transposing* the DataFrame:
  - One way to do this is simply to request the '`T`' parameter of the DataFrame:

```
~ print(df3.T)
>                    1242 1289 1308 1415 1417
> ID                    1    3    5    7    9
> Sex (Biological)      m    m    f    f    f
```

  - Alternatively, if you prefer functions, the Dataframe's '`transpose`' function will produce the same result:

```
~ print(df3.transpose())
>                    1242 1289 1308 1415 1417
> ID                    1    3    5    7    9
> Sex (Biological)      m    m    f    f    f
```

Naive Function Application

- In most cases Numpy functions can be applied directly to a DataFrame. These functions will be applied first column by column, then row-by-row, with the results retaining their index labels wherever possible:

```
~ # Calculate the sum of each column iteratively
~ result = np.sum(df3)
~ print(result)
> ID                        25
> Sex (Biological)    mmfff
> dtype: object
~ # Calculate the cumulative sum for each column
~ print(np.cumsum(df3))
>       ID Sex (Biological)
> 1242   1                 m
> 1289   4                mm
>              ...
> 1417  25             mmfff
```

The ‘apply’ Function

- For functions which require more than one parameter, we can use the ‘`apply`’ function. This requires 2 arguments, plus those we want to pass to the function for each application :
  - ‘*func*’: The function to apply to the DataFrame; the first of its argument that is not specified below will be what receives the row/column of the DataFrame.
  - ‘*axis*’: Whether the function should receive rows (`axis=1`) or columns (`axis=0`). If you only have one column/row, or want it to be applied cell by cell, this argument can be skipped
  - Any other arguments that are required for the function to run, labelled by their argument name within the original function

‘apply’ in Action

- Using a custom function which sets any value less than another specified value to 0, with the the other specified value being 5 for our ‘`apply`’:

```
~ # Function declaration
~ def omit_if_smaller(x, s):
~   if x < s:
~     return 0
~   else:
~     return x
~ # Applying the function to our ID value
~ print(df3.loc[:, "ID"].apply(func=omit_if_smaller, s=5)))
> 1242    0
>    ...
> 1417    9
> Name: ID, dtype: int64
```

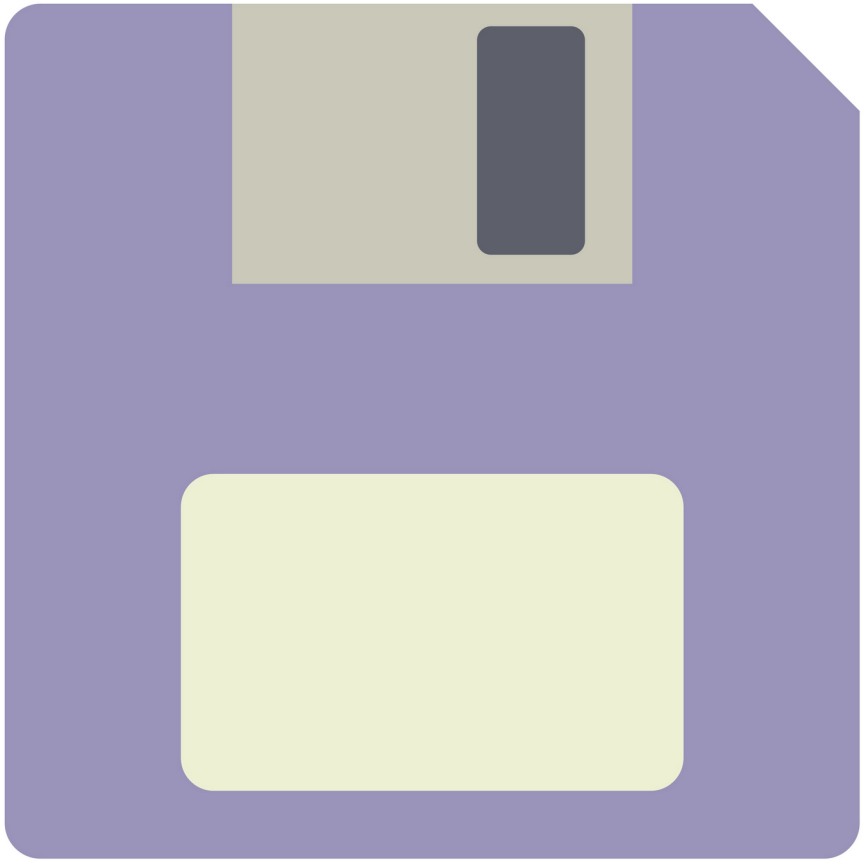
Saving your data

- We can save the data in a DataFrame to a '`.csv`' file using the '`to_csv`' function, which simply requires the filename with its extension:

```
~ df3.to_csv('output.csv')
```

- Likewise, the data can be saved to an excel spreadsheet ('`.xlsm`') using the '`to_excel`' function:

```
~
df3.to_excel('output.xlsm')
```



https://static.vecteezy.com/system/resources/previews/000/420/468/original/vector-save-icon.jpg

Putting it All Together!

UNIVERSITY OF CALGARY

Lets Test your Learning!

- You have been tasked again to create a Python program. This one should:
  - Load the data contained in '`iris_data.tsv`' into a DataFrame
  - Perform the following analyses on the data, reporting each to the user
    - Get the mean of Petal Width for all irises (`np.mean`)
    - Get the standard deviation of Petal Length for the 'Versicolour' species (`np.std`)
    - Identify which species has the largest observed Sepal Length in the dataset, without manual inspection! (`np.max`)

An incomplete Python script file has been provided with comment hints for each of the above steps for you to test your learning. A key for this document has also been provided, should you want to check your answer

Thank you!

Please send inquiries to kalum.ost@ucalgary.ca

UNIVERSITY OF CALGARY