Crash course on pandas

Slides by Sammy Klasfeld

(Please sign-in on the counter near the back door)

Penn
University of Pennsylvania

Sponsored by:

Institute for Biomedical Informatics

BGSA

GCB

Today's TAs

Apexa

Sammy

Ben

David

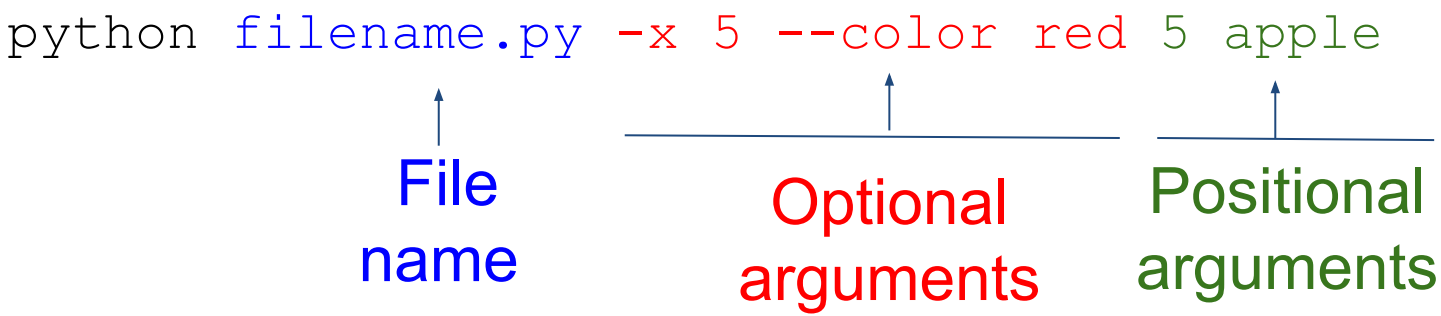
Will

Jake

Parisa

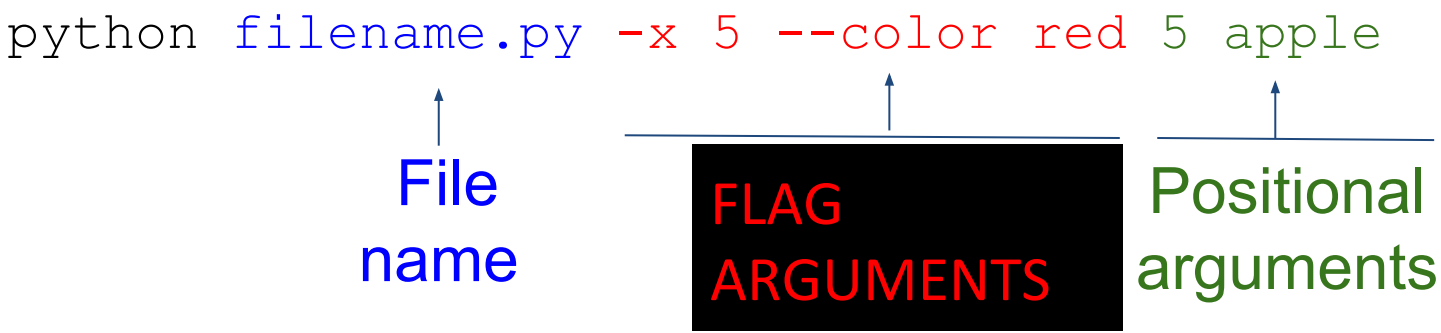
Argparse Comments (Lab 6)

- I made a mistake. In the last class, I distinguished two types of arguments:
  - Positional arguments
  - Optional arguments



Argparse Comments (Lab 6)

- I made a mistake. In the last class, I distinguished two types of arguments:
  - Positional arguments
  - ~~Optional arguments~~ Flag arguments:
    - Default to optional arguments, but can be set to required



Argparse Comments (Lab 6)

- There are two types of arguments: positional and flag

Argparse Comments (Lab 6)

- There are two types of arguments: positional and flag
- **Usually**, if you use argparse to organize your arguments in a script:
  - your flag arguments can go in any random order before or after the positional arguments.
  - You just need to make sure you put the positional arguments in the correct order.

Argparse argument order

These commands are all equal:

> python madlibs.py -a 42 3.14 -n Sheldon Karen **paper rock**

> python madlibs.py -n Sheldon Karen -a 42 3.14 **paper rock**

> python madlibs.py **paper rock** -a 42 3.14 -n Sheldon Karen

> python madlibs.py **paper rock** -n Sheldon Karen -a 42 3.14

These commands are all equal, but different from above:

> python madlibs.py -a 42 3.14 -n Sheldon Karen **rock paper**

> python madlibs.py -n Sheldon Karen -a 42 3.14 **rock paper**

> python madlibs.py **rock paper** -a 42 3.14 -n Sheldon Karen

> python madlibs.py **rock paper** -n Sheldon Karen -a 42 3.14

madlibs.py

```
import argparse

parser=argparse.ArgumentParser(description= \
        "fill in a madlib")
parser.add_argument('move_by_p1',
      choices=['rock', 'paper', 'scissors'],
      help="person1 choice of rock, paper, or scissors")
parser.add_argument('move_by_p2',
      choices=['rock', 'paper', 'scissors'],
      help="person2 choice of rock, paper, or scissors")
parser.add_argument("--names","-n", type=str,
        nargs=2, required=True, help="Two \
        peoples name. They cannot be the \
        same name.")
parser.add_argument("--ages","-a", type=int, nargs=2,
      required=False, help="Two peoples age.")
args=parser.parse_args()
```

Argparse Comments (Lab 6)

- There are two types of arguments: positional and flag
- **Usually**, if you use argparse to organize your arguments in a script:
  - your flag arguments can go in any random order before or after the positional arguments.
  - You just need to make sure you put the positional arguments in the correct order.
- WARNING: if you use nargs="*" or nargs="+" in either your positional or optional arguments, you may need to test your script to see which works.

Argparse argument order

Be careful if you set nargs to '*' or '+'

If you set nargs="*" in flag arguments you must put positional arguments before the flag aguments

```
> python test1.py -o1 op1 pos1 pos2
test1.py: error: the following arguments are required: pos_1, pos_2
```

The following works correctly:

```
> python test1.py pos1 pos2 -o1 op1
```

If you set **nargs="*" in positional arguments** things get tricky since python cannot distinguish between the position anymore which variable goes where. Therefore, if I want a list argument, I will use a flag argument and set `required=True`

test1.py

```
import argparse

parser=argparse.ArgumentParser()
parser.add_argument('pos_1', help="a string")
parser.add_argument('pos_2', help="a string")
parser.add_argument("--option_1","-o1", type=str,
        nargs="*", help="list of strings")
args=parser.parse_args()
```

What we need for today's class

Pandas and NumPy installed

Today's schedule

1. Introduction to Pandas
2. Creating a pandas dataframe
3. Selecting an index or column from a pandas dataframe
4. Writing pandas dataframes to a file
5. Updating DataFrames
    a. Creating new rows/columns
    b. Dropping rows/columns
    c. Sorting Columns/Rows
    d. Applying Functions to Columns/Rows/Values in a dataframe
6. Combining DataFrames together
7. Grouping Rows in DataFrames

1. Introduction to Pandas

What is pandas?

- Pandas is a python module that is capable of working with structured data fast, easy, and expressive
- Pandas introduced two data structures:
  - Series
  - DataFames

Series Data Structure

- Series are 1-dimensional data table
- You can think of a series as:
  - A list
  - A dictionary
  - A single row
  - A single column
  - An array

Series Data Structure

- Series are 1-dimensional data table
- You can think of a series as:
  - A list
  - A dictionary
  - A single row
  - A single column
  - An array

	Column A	**Column B**
Row1	<value_A1>	<value_A1>
Row2	<value_A2>	<value_A2>

Column A is an example of **a series with indexes**: [Row1, Row2].
Column B is another series with the same indexes.

Series Data Structure

- A series is a pandas 1D object

```
s= pd.Series([1,2,3,4,5])
```

- Contains an array of data: strings, floats, ints, etc
- Has an associated array of data labels called the **index** which correspond to the **values** in the series
  - By default the index is from `0` to `n-1`, where `n` is the length of the index (similar to a list)
  - However, the indexes can be set to strings (similar to keys in a dictionary)

Series Data Structure

```
# create series
import numpy as np
import pandas as pd
s = pd.Series(
    [5,3,6],
    ['a','b','c'])
print(s)
```

Here we set the **values** of the series to these numbers.

These are the **indexes** of our series

```
a   -0.039967
b    1.553495
c    1.909751
dtype: float64
```

Note that every series has a "dtype" which corresponds to the data-type of its values

Series Data Structure

Values in a series can be retrieved by either the location in the series (like a list) or their index value (like a dictionary)

```
>>> s = pd.Series([5,3,1],['a','b','c'])

>>> # return the first value of the series
>>> s[0]
5

>>> # return the value corresponding to index 'b'
>>> s['b']
3
```

Series Data Structure

Values in a series can be retrieved by either the location in the series (like a list) or their index value (like a dictionary)

```
>>> s = pd.Series([5,3,1],['a','b','c'])

>>> # return the first value of the series
>>> s[0]
5

>>> # return the value corresponding to index 'b'
>>> s['b']
3
```

Warning: if the items in the index are ints then items are only retrieved by their index values

```
>>> s = pd.Series([5,3,1],[1, 2, 3])
>>> s[1]
5
```

Series can be converted to/from lists or dictionaries

```
# create series from list
>>> list_x = [1,2,3]
>>> series_x = pd.Series(list_x)
>>> series_x
0   1
1   2
2   3
dtype: int64

# convert series to list
>>> series_x.to_list()
[1,2,3]
```

Series can be converted to/from lists or dictionaries

```
# create series from list
>>> list_x = [1,2,3]
>>> series_x = pd.Series(list_x)
>>> series_x
0   1
1   2
2   3
dtype: int64

# convert series to list
>>> series_x.to_list[]
[1,2,3]
```

```
# convert dictionary to series
>>> dict_y = {'a':1,'b':2,'c':3}
>>> series_y = pd.Series(dict_y)
>>> series_y
a   1
b   2
c   3
dtype: int64

# convert series to dictionary
>>> dict(series_y)
{'a': 1, 'b': 2, 'c': 3}
```

Functions with Pandas Series

```
print(s)# series
```

```
b  -1.197346
a   0.295999
c   0.094600
dtype: float64
```

```
# add series element-wise
print (s + s)
```

```
b  -2.394692
a   0.591998
c   0.189200
dtype: float64
```

```
# apply abs function
# to each item in series
s.apply(abs)
```

```
b  1.197346
a   0.295999
c   0.094600
dtype: float64
```

```
# get mean of total Series
s.mean()
```

```
-0.2689156801441579
```

A DataFrame is a Virtual Table

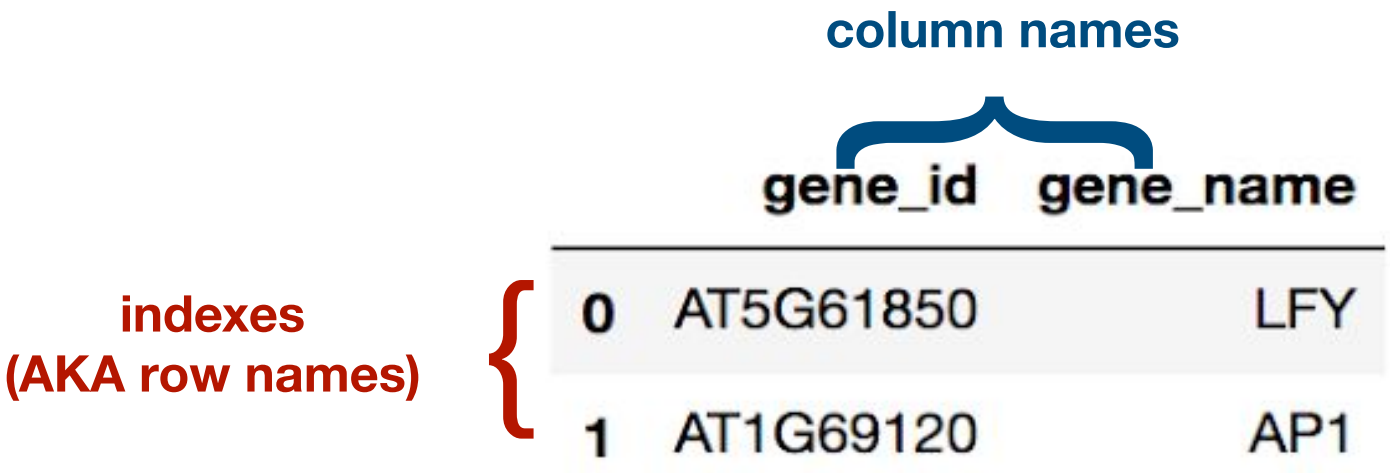
	Column A	Column B	Column C
Row 1	value or NaN	value or NaN	value or NaN
Row 2	value or NaN	value or NaN	value or NaN
Row 3	value or NaN	value or NaN	value or NaN
Row 4	value or NaN	value or NaN	value or NaN

DataFrame Syntax

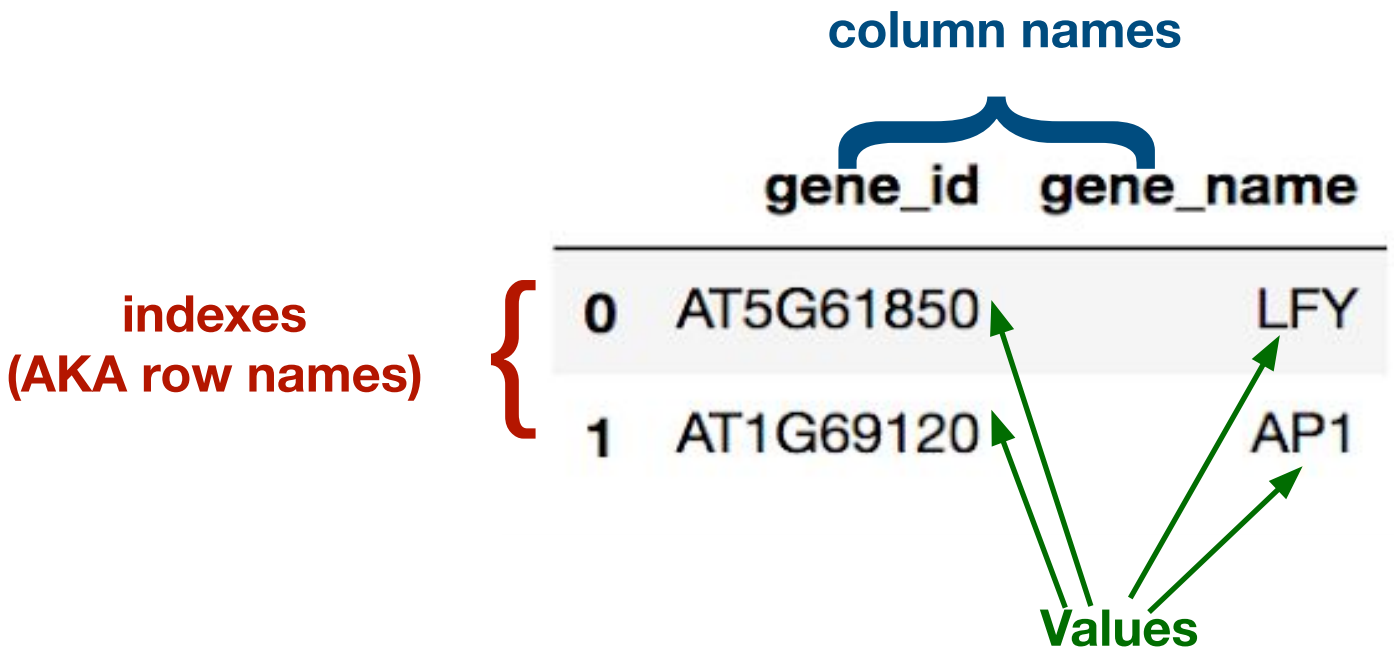
indexes (AKA row names)

	gene_id	gene_name
0	AT5G61850	LFY
1	AT1G69120	AP1

DataFrame Syntax



DataFrame Syntax



2. Creating a pandas DataFrame

(WARNING:There are many different way to create a pandas DataFrame)

We can create a DataFrame from a Dictionary of pandas series

```
1 dictOfSeries = {'treatmentA':pd.Series([2,4],['Billy','Bob']),
2                 'treatmentB':pd.Series([4,2],['Billy','Bob'])}
3 df = pd.DataFrame(dictOfSeries)
4 df
```

	treatmentA	treatmentB
Billy	2	4
Bob	4	2

Or we can create a DataFrame from a Dictionary of Lists

```
1  import pandas as pd # import pandas module
2
3  # create dictionary of lists
4  exp_dictOfList = {
5      "treatmentA":[2,4],
6      "treatmentB":[4,2]
7  }
8
9  # initialize the DataFrame
10 gene_frame = pd.DataFrame(exp_dictOfList,
11                            index=["Billy","Bob"]) # row names
12
13 gene_frame
```

	treatmentA	treatmentB
Billy	2	4
Bob	4	2

We can create a DataFrame from a List of Lists

```
1  import pandas as pd # import pandas module
2
3  # create list of lits
4  exp_listOflist = [
5      [2,4],
6      [4,2]
7  ]
8
9  # initialize the DataFrame
10 gene_frame = pd.DataFrame(exp_listOflist,
11                            columns = ["treatmentA","treatmentB"],
12                            index = ["Billy","Bob"]) # row names
13
14
15
16 gene_frame
```

	treatmentA	treatmentB
Billy	2	4
Bob	4	2

Or we can create a DataFrame from a List of Dictionaries

```
1  import pandas as pd # import pandas module
2
3  # create list of dictionaries
4  exp_listOfDict = [
5      {"treatmentA":2,"treatmentB":4},
6      {"treatmentA":4,"treatmentB":2}
7  ]
8
9  # initialize the DataFrame
10 gene_frame = pd.DataFrame(exp_listOfDict,
11                            index=["Billy","Bob"]) # row names
12
13
14
15 gene_frame
```

	treatmentA	treatmentB
Billy	2	4
Bob	4	2

We can create a DataFrame from a Dictionary of Dictionaries

```
1  import pandas as pd # import pandas module
2
3  # create dictionary of dictionary
4  exp_dictOfDict = {
5      "treatmentA":{"Billy":2, "Bob":4},
6      "treatmentB":{"Billy":4, "Bob":2}
7  }
8
9  # initialize the DataFrame
10 gene_frame = pd.DataFrame(exp_dictOfDict)
11
12 gene_frame
```

	treatmentA	treatmentB
Billy	2	4
Bob	4	2

Importing Pandas DataFrame from a file

1. Import pandas and numpy module
2. Write list of columns
3. Write dictionary of columns to their data-type
4. Read table into python compiler

Importing Pandas DataFrame from a file

For example, lets say we have this bed file “example.bed” filled with genes.
Note that a bed file is tab-delimited. In this case the columns are ordered as such:

1. Chromosome Location
2. Start Location
3. Stop Location
4. Name of Feature
5. Score (in this case this column is uninformative)
6. Feature Strand

example.bed

Chr1	1100	1600	GeneA	255	+
Chr2	15230	15342	GeneB	255	-
Chr2	15352	15455	GeneC	255	+
Chr2	17006	18110	GeneD	255	+
Chr3	10704	11671	GeneE	255	-

Step 1: Import Modules

```
1 # Step 1: Import modules
2 import pandas as pd # import pandas module
3 import numpy as np  # import numpy module
```

- We set pd as an alias for pandas and np as an alias for numpy now
- The NumPy module provides context for mathematical operations beyond the scope of the course

Step 2: Make a list of the column names

```
# Step 1: Import modules
import pandas as pd # import pandas module
import numpy as np  # import numpy module
```

```
# Step 2: List of Column Names
bed_columns = ["chr",     # chromosome location
               "start",   # start location
               "stop",    # stop location
               "name",    # name of the feature
               "score",   # score of feature
               "strand"] # feature strand
```

Step 3: Create a dictionary containing the data-type of each columns

(Otherwise python automatically classifies them and its NOT always right!)

```
# Step 1: Import modules
import pandas as pd # import pandas module
import numpy as np  # import numpy module
```

```
# Step 2: List of Column Names
bed_columns = ["chr",     # chromosome location
               "start",   # start location
               "stop",    # stop location
               "name",    # name of the feature
               "score",   # score of feature
               "strand"] # feature strand
```

```
# Step 3: Dictionary of Column Types
bedCol_types = {"chr": str, "start": np.int64,
                "stop": np.int64, "name": str,
                "score": np.float64,
                "strand": str}
```

Step 4: Read the table into Pandas!

```
# Step 4: Read the table into the python compiler
bed_df = pd.read_csv("example.bed",
                     sep = "\t",      # file is tab-delimited
                     header = None,   # the column names are
                                      # not in file
                     names = bedCols, # sets column names
                     dtype = bedCol_types) # sets column types
bed_df
```

	chr	start	stop	name	score	strand
0	Chr1	1100	1600	GeneA	255.0	+
1	Chr2	15230	15342	GeneB	255.0	-
2	Chr2	15352	15455	GeneC	255.0	+
3	Chr2	17006	18110	GeneD	255.0	+
4	Chr3	10704	11671	GeneE	255.0	-

For more parameters see : http://pandas.pydata.org/pandas-docs/stable/.

Pandas can even work with Excel!

See link:
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_excel.html

Basic DataFrame Functions

- bed_df**.head(n=*X*)** : gets the first ***X*** rows (default: n=5 if not stated)
- bed_df**.tail(n=*X*)** : gets the last ***X*** rows (default: n=5 if not stated)
- bed_df**.columns** : gets column names
- bed_df.**index** : gets row names
- bed_df.**dtypes** : gets datatypes of each column
- bed_df.**shape** : gets height and width of the DataFrame in a list format
- **len(**bed_df**)** : gets the height of the DataFrame
- bed_df.**describe**(): generates descriptive statistics

3. Selecting an index or column from a pandas dataframe

Access to columns

df =

	ONE	TWO
b	0.479859	-1.385437
a	0.123609	0.703162
c	0.491941	1.402770

- Access by attribute - can be converted to list via (df.ONE) function

```
1  df.ONE
```

```
b    0.479859
a    0.123609
c    0.491941
Name: ONE, dtype: float64
```

- Access by dictionary like notation - can be converted to list via list(df[‘ONE’]) function

```
1  df["TWO"]
```

```
b   -1.385437
a    0.703162
c    1.402770
Name: TWO, dtype: float64
```

Selecting an Index or Column From a DataFrame using .iloc & .loc

- .iloc[<row_idx>, <col_idx>] - command to get values in DataFrame based on row and column locations

- .loc[<row_names>, <col_names>] - command to get values in DataFrame based on row values and column values

Which rows will print after the following command (from which row indexes)?

```
1 import pandas as pd
2 peoples_df = pd.DataFrame(
3     {"name" : ["Bob", "Sue", "Tom", "Pam"],
4      "age" : [56, 21, 22, 35],
5      "inches" : [71, 63.5, 68.5, 62]})
6 peoples_df
```

	age	inches	name
0	56	71.0	Bob
1	21	63.5	Sue
2	22	68.5	Tom
3	35	62.0	Pam

```
1 peoples_df.iloc[range(1,3),:]
```

Which rows will print after the following command (from which row indexes)?

```
1 import pandas as pd
2 peoples_df = pd.DataFrame(
3     {"name" : ["Bob", "Sue", "Tom", "Pam"],
4      "age" : [56, 21, 22, 35],
5      "inches" : [71, 63.5, 68.5, 62]})
6 peoples_df
```

	age	inches	name
0	56	71.0	Bob
1	21	63.5	Sue
2	22	68.5	Tom
3	35	62.0	Pam

```
1 peoples_df.iloc[range(1,3),:]
```

variable name for DataFrame

Which rows will print after the following command (from which row indexes)?

```
1 import pandas as pd
2 peoples_df = pd.DataFrame(
3     {"name" : ["Bob", "Sue", "Tom", "Pam"],
4      "age" : [56, 21, 22, 35],
5      "inches" : [71, 63.5, 68.5, 62]})
6 peoples_df
```

	age	inches	name
0	56	71.0	Bob
1	21	63.5	Sue
2	22	68.5	Tom
3	35	62.0	Pam

```
1 peoples_df.iloc[range(1,3),:]
```

variable name for DataFrame

get values in DataFrame based on row and column locations

Which rows will print after the following command (from which row indexes)?

```
1 import pandas as pd
2 peoples_df = pd.DataFrame(
3     {"name" : ["Bob", "Sue", "Tom", "Pam"],
4      "age" : [56, 21, 22, 35],
5      "inches" : [71, 63.5, 68.5, 62]})
6 peoples_df
```

	age	inches	name
0	56	71.0	Bob
1	21	63.5	Sue
2	22	68.5	Tom
3	35	62.0	Pam

This gives a range of numbers from 1-2 for row locations

```
1 peoples_df.iloc[range(1,3),:]
```

variable name for DataFrame

get values in DataFrame based on row and column locations

Which rows will print after the following command (from which row indexes)?

```
1 import pandas as pd
2 peoples_df = pd.DataFrame(
3     {"name" : ["Bob", "Sue", "Tom", "Pam"],
4      "age" : [56, 21, 22, 35],
5      "inches" : [71, 63.5, 68.5, 62]})
6 peoples_df
```

	age	inches	name
0	56	71.0	Bob
1	21	63.5	Sue
2	22	68.5	Tom
3	35	62.0	Pam

This gives a range of numbers from 1-2 for row locations

```
1 peoples_df.iloc[range(1,3),:]
```

variable name for DataFrame

get values in DataFrame based on row and column locations

This colon symbol represents all of the columns

Which rows will print after the following command (from which row indexes)?

```
1 import pandas as pd
2 peoples_df = pd.DataFrame(
3     {"name" : ["Bob", "Sue", "Tom", "Pam"],
4      "age" : [56, 21, 22, 35],
5      "inches" : [71, 63.5, 68.5, 62]})
6 peoples_df
```

	age	inches	name
0	56	71.0	Bob
1	21	63.5	Sue
2	22	68.5	Tom
3	35	62.0	Pam

```
1 peoples_df.iloc[range(1,3),:]
```

	age	inches	name
1	21	63.5	Sue
2	22	68.5	Tom

ANSWER: This prints rows with indexes 1 & 2 since they are the second and third row

Which rows will print after the following command?

```
1 peoples_df
```

	age	inches	name
0	56	71.0	Bob
1	21	63.5	Sue
2	22	68.5	Tom
3	35	62.0	Pam

```
1 peoples_df.loc[range(1,3),:]
```

Which rows will print after the following command?

```
1 peoples_df
```

	age	inches	name
0	56	71.0	Bob
1	21	63.5	Sue
2	22	68.5	Tom
3	35	62.0	Pam

```
1 peoples_df.loc[range(1,3),:]
```

	age	inches	name
1	21	63.5	Sue
2	22	68.5	Tom

ANSWER: This also prints rows with indexes 1 & 2 since the value of these row indexes are 1 & 2 respectively since we do NOT have row names

Setting Row Names with **.set_index()**

```
1 peoples_df
```

	age	inches	name
0	56	71.0	Bob
1	21	63.5	Sue
2	22	68.5	Tom
3	35	62.0	Pam

```
1 # set the row names to the "name" column
2 peoples_df = peoples_df.set_index("name")
3 peoples_df
```

	age	inches
name		
Bob	56	71.0
Sue	21	63.5
Tom	22	68.5
Pam	35	62.0

Which rows will print after the following command?

```
1 peoples_df
```

name	age	inches
Bob	56	71.0
Sue	21	63.5
Tom	22	68.5
Pam	35	62.0

```
1 peoples_df.loc[1,:]
```

Which rows will print after the following command?

```
1 peoples_df
```

name	age	inches
Bob	56	71.0
Sue	21	63.5
Tom	22	68.5
Pam	35	62.0

```
1 peoples_df.loc[1,:]
```

ANSWER: This prints an ERROR since there is no row name “1”

Which rows will print after the following command?

```
1 peoples_df
```

name	age	inches
Bob	56	71.0
Sue	21	63.5
Tom	22	68.5
Pam	35	62.0

```
1 peoples_df.loc[["Bob","Tom"],:]
```

Which rows will print after the following command?

```
1 peoples_df
```

name	age	inches
Bob	56	71.0
Sue	21	63.5
Tom	22	68.5
Pam	35	62.0

```
1 peoples_df.loc[["Bob","Tom"],:]
```

name	age	inches
Bob	56	71.0
Tom	22	68.5

ANSWER: This prints the rows with row names “Bob” and “Tom”

What value would this print?

```
1 peoples_df
```

name	age	inches
Bob	56	71.0
Sue	21	63.5
Tom	22	68.5
Pam	35	62.0

```
1 peoples_df.iloc[1,1]
```

What value would this print?

```
1 peoples_df
```

name	age	inches
Bob	56	71.0
Sue	21	63.5
Tom	22	68.5
Pam	35	62.0

```
1 peoples_df.iloc[1,1]
```

ANSWER: 63.5

What value would this print?

```
1 peoples_df
```

name	age	inches
Bob	56	71.0
Sue	21	63.5
Tom	22	68.5
Pam	35	62.0

```
1 peoples_df.loc["Sue","age"]
```

What value would this print?

```
1 peoples_df
```

name	age	inches
Bob	56	71.0
Sue	21	63.5
Tom	22	68.5
Pam	35	62.0

```
1 peoples_df.loc["Sue","age"]
```

ANSWER: 21

Which column will print after the following command?

```
1 peoples_df
```

name	age	inches
Bob	56	71.0
Sue	21	63.5
Tom	22	68.5
Pam	35	62.0

```
1 peoples_df.loc[:,"age"]
```

Which column will print after the following command?

```
1 peoples_df
```

name	age	inches
Bob	56	71.0
Sue	21	63.5
Tom	22	68.5
Pam	35	62.0

```
1 peoples_df.loc[:,"age"]
```

```
name
Bob     56
Sue     21
Tom     22
Pam     35
Name: age, dtype: int64
```

ANSWER: The “age” column

But why is the output formatted differently?

DataFrame Conditionals

- Occasionally you may want to subset a DataFrame based on specific columns. For example, we may want to subset the “peoples_df” to people under the age of 50.
- Using the .loc command we can use conditionals on rows (the first parameter) and/or columns (the second parameter)

```
1  peoples_df.loc[peoples_df["age"]<50,:]
```

name	age	inches
Sue	21	63.5
Tom	22	68.5
Pam	35	62.0

What value would this print?

```
1 peoples_df
```

name	age	inches
Bob	56	71.0
Sue	21	63.5
Tom	22	68.5
Pam	35	62.0

```
1 min(peoples_df.loc[peoples_df.loc[:,"age"]<50,"inches"])
```

What value would this print?

```
1 peoples_df
```

name	age	inches
Bob	56	71.0
Sue	21	63.5
Tom	22	68.5
Pam	35	62.0

```
1 min(peoples_df.loc[peoples_df.loc[:,"age"]<50,"inches"])
```

ANSWER: 62.0

What row(s) would this print?

```
1  peoples_df
```

name	age	inches
Bob	56	71.0
Sue	21	63.5
Tom	22	68.5
Pam	35	62.0

Note: for multiple conditions inside .loc or .iloc we only use a single ampersand

```
1  peoples_df.loc[(peoples_df.loc[:,"age"]<50) &
2                 (peoples_df.loc[:,"inches"]>65),:]
```

What row(s) would this print?

```
1  peoples_df
```

name	age	inches
Bob	56	71.0
Sue	21	63.5
Tom	22	68.5
Pam	35	62.0

```
1  peoples_df.loc[(peoples_df.loc[:,"age"]<50) &
2                 (peoples_df.loc[:,"inches"]>65),:]
```

ANSWER:

name		
Tom	22	68.5

4. Writing a dataframe to a file

Exporting a DataFrame to a File

Similar to importing a text file to a *pandas* DataFrame, there are also several parameters for exporting a pandas DataFrame.

The syntax for exporting a DataFrame is :

```
dataFrame.to_csv(path_to_outputFile, arguments)
```

For example, to export a tab-delimited file from DataFrame `test_df` with a header and no row-names to the file “test.txt” we use the command:

```
test_df.to_csv("test.txt", sep="\t", header=True, index=False)
```

5. Updating DataFrames

Creating New Columns

You should always add columns using the .loc and .iloc commands to avoid SettingWithCopyWarnings.

For example, we can add a row for height in “centimeters” in addition to inches:

```
peoples_df.loc[:,"centimeters"] = \
    Peoples_df.loc[:,"inches"] * 2.54
peoples_df
```

name	age	inches	centimeters
Bob	56	71.0	180.34
Sue	21	63.5	161.29
Tom	22	68.5	173.99
Pam	35	62.0	157.48

Dropping Rows/Columns

Drop rows or columns using the .drop() function

```
peoples_df
```

name	age	inches	centimeters
Bob	56	71.0	180.34
Sue	21	63.5	161.29
Tom	22	68.5	173.99
Pam	35	62.0	157.48

```
# drop row from peoples table
peoples_df.drop(["Tom"])
```

name	age	inches	centimeters
Bob	56	71.0	180.34
Sue	21	63.5	161.29
Pam	35	62.0	157.48

Dropping Rows/Columns

Drop rows or columns using the .drop() function

```
peoples_df
```

name	age	inches	centimeters
Bob	56	71.0	180.34
Sue	21	63.5	161.29
Tom	22	68.5	173.99
Pam	35	62.0	157.48

```
# drop row from peoples table
peoples_df.drop(["Tom"])
```

name	age	inches	centimeters
Bob	56	71.0	180.34
Sue	21	63.5	161.29
Pam	35	62.0	157.48

```
# drop column from peoples table
peoples_df.drop(["inches"],axis=1)
```

name	age	centimeters
Bob	56	180.34
Sue	21	161.29
Tom	22	173.99
Pam	35	157.48

Dropping Rows/Columns

Drop rows or columns using the .drop() function

```
peoples_df
```

name	age	inches	centimeters
Bob	56	71.0	180.34
Sue	21	63.5	161.29
Tom	22	68.5	173.99
Pam	35	62.0	157.48

```
# drop row from peoples table
peoples_df.drop(["Tom"])
```

name	age	inches	centimeters
Bob	56	71.0	180.34
Sue	21	63.5	161.29
Pam	35	62.0	157.48

Note: This function DOES NOT RUN IN PLACE. This means that we must set our dataframe to our command to update it

```
# drop column from peoples table
peoples_df.drop(["inches"],axis=1)
```

name	age	centimeters
Bob	56	180.34
Sue	21	161.29
Tom	22	173.99
Pam	35	157.48

Sorting DataFrames

Sort rows using the .sort_values() function

```
# original table
peoples_df
```

name	age	inches	centimeters
Bob	56	71.0	180.34
Sue	21	63.5	161.29
Tom	22	68.5	173.99
Pam	35	62.0	157.48

```
# sort row by column "inches"
peoples_df.sort_values("inches",ascending=True)
```

name	age	inches	centimeters
Pam	35	62.0	157.48
Sue	21	63.5	161.29
Tom	22	68.5	173.99
Bob	56	71.0	180.34

- This function also DOES NOT RUN IN PLACE. This means that we must set our dataframe to our command update it.
- By default, .sort_values() sorts rows, but if user sets “axis=1” inside the function, it will sort columns instead.

Applying functions to DataFrames

```
1 df
```

	ONE	TWO
b	0.639437	1.599297
a	0.401695	-1.267588
c	0.313774	-1.120311

```
1 # get the absolute values
2 # of each value in df
3 df.apply(np.abs)
```

	ONE	TWO
b	0.639437	1.599297
a	0.401695	1.267588
c	0.313774	1.120311

Applying functions to DataFrames

```
1  df
```

	ONE	TWO
b	0.639437	1.599297
a	0.401695	-1.267588
c	0.313774	-1.120311

```
1  # get the absolute values
2  # of each value in df
3  df.apply(np.abs)
```

	ONE	TWO
b	0.639437	1.599297
a	0.401695	1.267588
c	0.313774	1.120311

```
1  # get the sum of each column
2  df.apply(np.sum, axis=1)
```

```
b    2.238734
a   -0.865893
c   -0.806537
dtype: float64
```

Applying functions to DataFrames

```
1 df
```

	ONE	TWO
b	0.639437	1.599297
a	0.401695	-1.267588
c	0.313774	-1.120311

```
1 # get the absolute values
2 # of each value in df
3 df.apply(np.abs)
```

	ONE	TWO
b	0.639437	1.599297
a	0.401695	1.267588
c	0.313774	1.120311

```
1 # get the sum of each column
2 df.apply(np.sum, axis=1)
```

```
b    2.238734
a   -0.865893
c   -0.806537
dtype: float64
```

```
1 # get the mean of each r0w
2 df.apply(np.mean, axis=0)
```

```
ONE    0.451635
TWO   -0.262867
dtype: float64
```

Applying functions to DataFrames

```
df
```

	ONE	TWO
b	0.639437	1.599297
a	0.401695	-1.267588
c	0.313774	-1.120311

```
# get the absolute values
# of each value in df
df.apply(np.abs)
```

	ONE	TWO
b	0.639437	1.599297
a	0.401695	1.267588
c	0.313774	1.120311

```
# get the sum of each column
df.apply(np.sum, axis=1)
```

```
b    2.238734
a   -0.865893
c   -0.806537
dtype: float64
```

```
# get the mean of each r0w
df.apply(np.mean, axis=0)
```

```
ONE    0.451635
TWO   -0.262867
dtype: float64
```

```
# subtract 1 from each value
# using a custom function
df.apply(lambda x: x-1)
```

	ONE	TWO
b	-0.360563	0.599297
a	-0.598305	-2.267588
c	-0.686226	-2.120311

6. Combining DataFrames together

Combining DataFrames

Up to this point, we have been working with a single table at a time, but most of the time we work with many tables that can be related to each other in some way.

Concatenating DataFrames

It is easy to merge **multiple dataframes that have the same columns** together using the pandas concat() function

```
1  import numpy as np
2  import pandas as pd
3  s1 = pd.Series(
4      np.random.randn(3))
5  d1 = {'ONE':s1*s1, 'TWO': s1+s1}
6  df1 = pd.DataFrame(d1)
7  s2 = pd.Series(
8      np.random.randn(3))
9  d2 = {'ONE':s2*s2, 'TWO': s2+s2}
10 df2 = pd.DataFrame(d2)
```

```
1  df1
```

	ONE	TWO
0	0.181987	0.853200
1	0.124461	-0.705579
2	0.079081	0.562427

```
1  df2
```

	ONE	TWO
0	0.010352	-0.203493
1	0.036690	0.383092
2	0.136860	-0.739893

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

	ONE	TWO
0	0.181987	0.853200
1	0.124461	-0.705579
2	0.079081	0.562427
0	0.010352	-0.203493
1	0.036690	0.383092
2	0.136860	-0.739893

```
1  pd.concat([df1,df2], ignore_index=True)
```

	ONE	TWO
0	0.181987	0.853200
1	0.124461	-0.705579
2	0.079081	0.562427
3	0.010352	-0.203493
4	0.036690	0.383092
5	0.136860	-0.739893

We use **ignore_index** to reset the indices so that there are none that are redundant

Merging DataFrames

- Concatenation is useful if your data frames have the same types of information (the same columns) but about different identities (different rows)
- Sometimes you have tables that contain data about the same set of items (found in rows) but contain different information about those items (some different columns)

Peak ID	Gene ID	Distance From Peak to Gene
1	AT5G61850	50
2	AT5G61850	0
3	AT1G17180	0
4	AT2G11120	210

Gene ID	Function
AT1G17180	Ribosomal
AT1G17190	Respiratory
AT1G17200	Respiratory
AT1G17201	Stress Response

Database Terminology

- **Primary key** - column(s) in a table that is a unique identifier for each row
- **Foreign key** - column(s) in a table that specify a link to a primary key in another table
- These keys are where we combine two tables

Foreign Key

Peak ID	Gene ID	Distance From Peak to Gene
1	AT5G61850	50
2	AT5G61850	0
3	AT1G17180	0
4	AT2G11120	210

Primary Key

Gene ID	Function
AT1G17180	Ribosomal
AT1G17190	Respiratory
AT1G17200	Respiratory
AT1G17201	Stress Response

Pandas merge function

```
>>>df1 = pd.DataFrame({'lkey': ['foo', 'bar', 'baz', 'faa'],
                     'value': [1, 2, 3, 5]})
>> df1
   lkey value
0   foo  1
1   bar  2
2   baz  3
3   faa  5
```

```
>>>df2 = pd.DataFrame({'rkey': ['foo', 'bar', 'baz', 'faa'],
                     'value': [1, 2, 3, 5]})
>> df2
   rkey value
0   foo  6
1   bar  7
2   baz  8
3   faa  9
```

```
df1.merge(df2, left_on='lkey', right_on='rkey')
```

Left data frame

Right data frame

Left primary/ foreign key

Right primary/ foreign key

Pandas merge function

```
>>>df1 = pd.DataFrame({'lkey': ['foo', 'bar', 'baz', 'faa'],
                     'value': [1, 2, 3, 5]})
>> df1
```

	lkey	value
0	foo	1
1	bar	2
2	baz	3
3	faa	5

```
>>>df2 = pd.DataFrame({'rkey': ['foo', 'bar', 'baz', 'faa'],
                     'value': [1, 2, 3, 5]})
>> df2
```

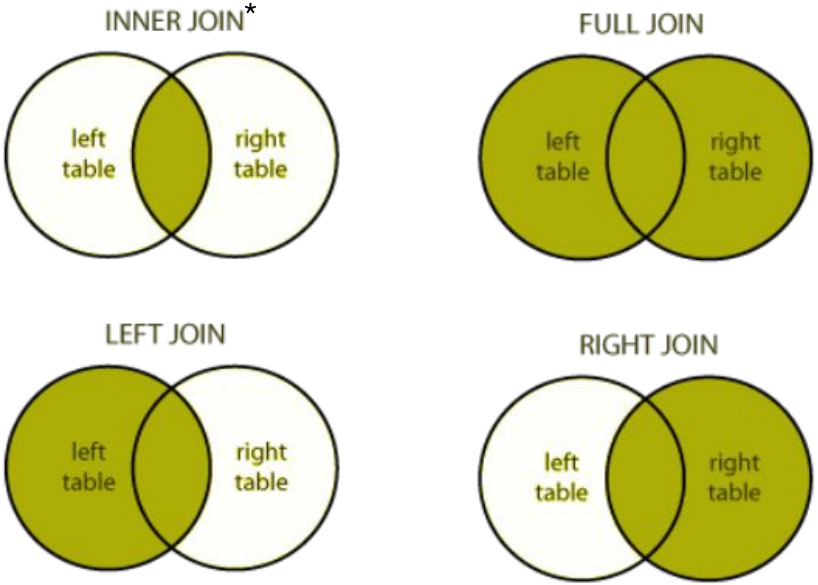
	rkey	value
0	foo	6
1	bar	7
2	baz	8
3	faa	9

```
df1.merge(df2, left_on='lkey', right_on='rkey')
```

	lkey	value_x	rkey	value_y
0	foo	1	foo	5
1	bar	2	bar	6
2	baz	3	baz	7
3	faa	5	faa	8

Different Ways to join a DataFrame

- Not all data frames have the exact same foreign keys that match the primary keys. Some keys should match but each data frame may have keys that are not found in the other.
- There are four primary ways to join two tables together



* INNER JOIN is the default for pandas merge

Different Ways to join a DataFrame

IDs in df1: a,b,c IDs in df2: a,b,d

Different Ways to join a DataFrame

inner join

IDs in df1: a,b,c IDs in df2: a,b,d ----> IDs in merged: a,b

Different Ways to join a DataFrame

inner join

IDs in df1: a,b,c IDs in df2: a,b,d ----> IDs in merged: a,b

left join

IDs in df1: a,b,c IDs in df2: a,b,d ----> IDs in merged: a,b,c

Different Ways to join a DataFrame

inner join

IDs in df1: a,b,c IDs in df2: a,b,d ----> IDs in merged: a,b

left join

IDs in df1: a,b,c IDs in df2: a,b,d ----> IDs in merged: a,b,c

right join

IDs in df1: a,b,c IDs in df2: a,b,d ----> IDs in merged: a,b,d

Different Ways to join a DataFrame

inner join

IDs in df1: a,b,c IDs in df2: a,b,d ----> IDs in merged: a,b

left join

IDs in df1: a,b,c IDs in df2: a,b,d ----> IDs in merged: a,b,c

right join

IDs in df1: a,b,c IDs in df2: a,b,d ----> IDs in merged: a,b,d

outer join

IDs in df1: a,b,c IDs in df2: a,b,d ----> IDs in merged: a,b,c,d

Game Time:
NAME THAT JOIN

WHAT JOIN DID WE USE?

student_df

	student_id	first_name	last_name
0	111	Alex	Anderson
1	222	Amy	Brown
2	333	Alan	Cooper

course_df

	sid	course_name
0	111	Calculus
1	111	Chemistry
2	222	Biology
3	444	English

AFTER MERGE

	student_id	first_name	last_name	sid	course_name
0	111	Alex	Anderson	111	Calculus
1	111	Alex	Anderson	111	Chemistry
2	222	Amy	Brown	222	Biology
3	333	Alan	Cooper	NaN	Nan
4	NaN	NaN	NaN	444	English

WHAT JOIN DID WE USE?

student_df

	student_id	first_name	last_name
0	111	Alex	Anderson
1	222	Amy	Brown
2	333	Alan	Cooper

course_df

	sid	course_name
0	111	Calculus
1	111	Chemistry
2	222	Biology
3	444	English

AFTER MERGE

	student_id	first_name	last_name	sid	course_name
0	111	Alex	Anderson	111	Calculus
1	111	Alex	Anderson	111	Chemistry
2	222	Amy	Brown	222	Biology
3	333	Alan	Cooper	NaN	Nan
4	NaN	NaN	NaN	444	English

OUTER JOIN!

```
student_df.merge(
    course_df,
    left_on='student_id',
    right_on='sid',
    how='outer')
```

WHAT JOIN DID WE USE?

student_df

	student_id	first_name	last_name
0	111	Alex	Anderson
1	222	Amy	Brown
2	333	Alan	Cooper

course_df

	sid	course_name
0	111	Calculus
1	111	Chemistry
2	222	Biology
3	444	English

AFTER MERGE

	student_id	first_name	last_name	sid	course_name
0	111	Alex	Anderson	111	Calculus
1	111	Alex	Anderson	111	Chemistry
2	222	Amy	Brown	222	Biology
4	NaN	NaN	NaN	444	English

WHAT JOIN DID WE USE?

student_df

	student_id	first_name	last_name
0	111	Alex	Anderson
1	222	Amy	Brown
2	333	Alan	Cooper

course_df

	sid	course_name
0	111	Calculus
1	111	Chemistry
2	222	Biology
3	444	English

AFTER MERGE

	student_id	first_name	last_name	sid	course_name
0	111	Alex	Anderson	111	Calculus
1	111	Alex	Anderson	111	Chemistry
2	222	Amy	Brown	222	Biology
4	NaN	NaN	NaN	444	English

RIGHT JOIN!

```
student_df.merge(
    course_df,
    left_on='student_id',
    right_on='sid',
    how='outer')
```

WHAT JOIN DID WE USE?

student_df

	student_id	first_name	last_name
0	111	Alex	Anderson
1	222	Amy	Brown
2	333	Alan	Cooper

course_df

	sid	course_name
0	111	Calculus
1	111	Chemistry
2	222	Biology
3	444	English

AFTER MERGE

	student_id	first_name	last_name	sid	course_name
0	111	Alex	Anderson	111	Calculus
1	111	Alex	Anderson	111	Chemistry
2	222	Amy	Brown	222	Biology

WHAT JOIN DID WE USE?

student_df

	student_id	first_name	last_name
0	111	Alex	Anderson
1	222	Amy	Brown
2	333	Alan	Cooper

course_df

	sid	course_name
0	111	Calculus
1	111	Chemistry
2	222	Biology
3	444	English

AFTER MERGE

	student_id	first_name	last_name	sid	course_name
0	111	Alex	Anderson	111	Calculus
1	111	Alex	Anderson	111	Chemistry
2	222	Amy	Brown	222	Biology

INNER JOIN!

```
student_df.merge(
    course_df,
    left_on='student_id',
    right_on='sid')
```

WHAT JOIN DID WE USE?

student_df

	student_id	first_name	last_name
0	111	Alex	Anderson
1	222	Amy	Brown
2	333	Alan	Cooper

course_df

	sid	course_name
0	111	Calculus
1	111	Chemistry
2	222	Biology
3	444	English

AFTER MERGE

	student_id	first_name	last_name	sid	course_name
0	111	Alex	Anderson	111	Calculus
1	111	Alex	Anderson	111	Chemistry
2	222	Amy	Brown	222	Biology
3	333	Alan	Cooper	NaN	Nan

WHAT JOIN DID WE USE?

student_df

	student_id	first_name	last_name
0	111	Alex	Anderson
1	222	Amy	Brown
2	333	Alan	Cooper

course_df

	sid	course_name
0	111	Calculus
1	111	Chemistry
2	222	Biology
3	444	English

AFTER MERGE

	student_id	first_name	last_name	sid	course_name
0	111	Alex	Anderson	111	Calculus
1	111	Alex	Anderson	111	Chemistry
2	222	Amy	Brown	222	Biology
3	333	Alan	Cooper	NaN	Nan

LEFT JOIN!

```
student_df.merge(
    course_df,
    left_on='student_id',
    right_on='sid',
    how='left')
```

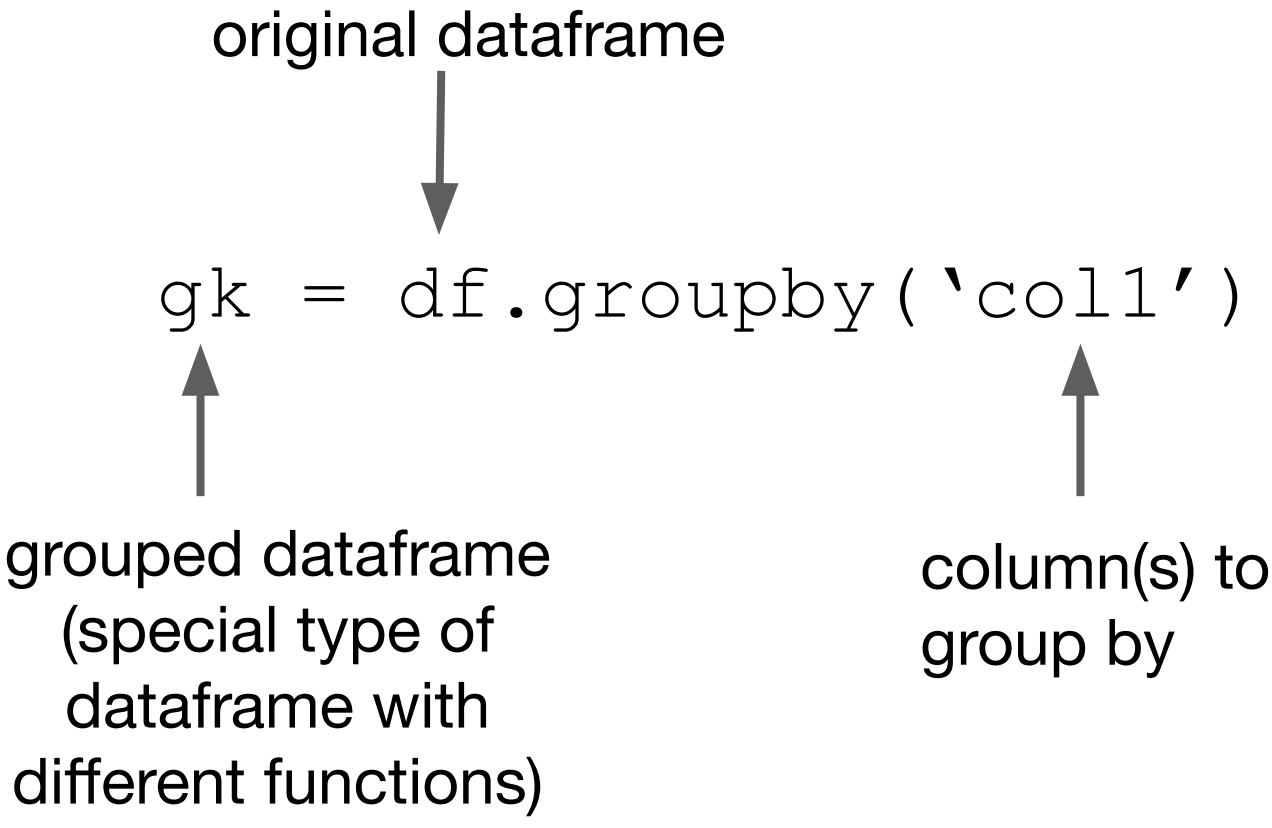
7. Grouping pandas dataframes

What is grouping?

- Grouping means to split a data frame into groups based on some criteria.
- For example, if I have a list of ChIP-seq peaks that I annotated to genes. I may want to group the peaks on the genes they are annotated to.

	peak_name	**gene_name**
0	peak1	geneA
1	peak2	geneB
2	peak3	geneC
3	peak4	geneC

dataframe.groupby() function



dataframe.groupby() function

	peak	gene
0	peak1	geneA
1	peak2	geneB
2	peak3	geneC
3	peak4	geneC
4	peak5	geneC
5	peak6	geneE
6	peak7	geneE
7	peak8	geneH
8	peak9	geneH

`df.groupby('gene')`

Besides grouping, This is also called “splitting” the dataframe

	peak	gene
0	peak1	geneA

	peak	gene
1	peak2	geneB

	peak	gene
2	peak3	geneC
3	peak4	geneC
4	peak5	geneC

	peak	gene
5	peak6	geneE
6	peak7	geneE

	peak	gene
7	peak8	geneH
8	peak9	geneH

Looping through grouped Data Frames

- The format for looping through grouped Data Frames is:

```
for <grouping_col>,<sub_df> in <grouped_df>
```

Looping through grouped Data Frames

```
1 peak_gene_df
```

	peak	gene
0	peak1	geneA
1	peak2	geneB
2	peak3	geneC
3	peak4	geneE
4	peak5	geneE
5	peak6	geneC
6	peak7	geneC

```
1 peak_gene_grouped = peak_gene_df.groupby("gene")
2 for gene_name, gene_df in peak_gene_grouped:
3   print(gene_name)
4   print(gene_df)
```

```
geneA
    peak1   geneA
geneB
    peak2   geneB
geneC
    peak3   geneC
    peak6   geneC
    peak7   geneC
geneE
    peak4   geneE
    peak5   geneE
```

Access Specific Groups

```
peak_gene_df
```

	peak	gene	log10_pvalue
0	peak1	geneA	30
1	peak2	geneB	10
2	peak3	geneC	11
3	peak4	geneE	15
4	peak5	geneE	23
5	peak6	geneC	99
6	peak7	geneC	102

```
# get dataframe for geneC only
peak_gene_grouped = \
  peak_gene_df.groupby("gene")
peak_gene_grouped.get_group("geneE")
```

	peak	gene	log10_pvalue
2	peak3	geneC	11
5	peak6	geneC	99
6	peak7	geneC	102

Perform Functions on Groups

(split, apply,combine)

```
peak_gene_df
```

	peak	gene	log10_pvalue
0	peak1	geneA	30
1	peak2	geneB	10
2	peak3	geneC	11
3	peak4	geneE	15
4	peak5	geneE	23
5	peak6	geneC	99
6	peak7	geneC	102

```
# apply a max function to each group
peak_gene_grouped = \
   peak_gene_df.groupby("gene") # split
# apply & combine
peak_gene_grouped["log10_pvalues"].apply(np.max)
```

	log10_pvalue
gene	
geneA	30
geneB	10
geneC	102
geneE	23

Summary

- We can now build pandas DataFrames in python using lists, dictionaries, or files
- We can subset these DataFrames
- We can export these DataFrames to a File
- We can sort and apply functions to DataFrames
- Merge ataFrames together using the .join() function
- Group series together using .groupby() function

Next Lecture

- Visualization and plotnine
- Make sure you have plotnine installed

APPENDIX

Functions to deal with missing data

Fill the missing data

```
>> df
        one       two     three four   five  timestamp
a       NaN -0.282863 -1.509059  bar   True        NaT
c       NaN  1.212112 -0.173215  bar  False        NaT
e  0.119209 -1.044236 -0.861849  bar   True 2012-01-01
f -2.104569 -0.494929  1.071804  bar  False 2012-01-01

>> df.fillna(0) # fill missing data with 0's
       one       two     three four   five   timestamp
a  0.000000 -0.282863 -1.509059  bar   True  0
c  0.000000  1.212112 -0.173215  bar  False  0
e  0.119209 -1.044236 -0.861849  bar   True  2012-01-01 00:00:00
f -2.104569 -0.494929  1.071804  bar  False  2012-01-01 00:00:00
```

Drop axis labels with missing data

drop ROWS if ANY column contains an empty value

```
>> df
        one       two     three four   five  timestamp
a       NaN -0.282863 -1.509059  bar   True        NaT
c       NaN  1.212112 -0.173215  bar  False        NaT
e  0.119209 -1.044236 -0.861849  bar   True 2012-01-01
f -2.104569 -0.494929  1.071804  bar  False 2012-01-01

>> df.dropna()
       one       two     three four   five   timestamp
e  0.119209 -1.044236 -0.861849  bar   True  2012-01-01 00:00:00
f -2.104569 -0.494929  1.071804  bar  False  2012-01-01 00:00:00
```

Drop axis labels with missing data

drop COLUMNS if ANY column contains an empty value

```
>> df
        one       two     three four   five  timestamp
a       NaN -0.282863 -1.509059  bar   True        NaT
c       NaN  1.212112 -0.173215  bar  False        NaT
e  0.119209 -1.044236 -0.861849  bar   True 2012-01-01
f -2.104569 -0.494929  1.071804  bar  False 2012-01-01

>> df.dropna(axis="columns")
       one       two     three four   five   timestamp
a  0.000000 -0.282863 -1.509059  bar   True  0
c  0.000000  1.212112 -0.173215  bar  False  0
e  0.119209 -1.044236 -0.861849  bar   True  2012-01-01 00:00:00
f -2.104569 -0.494929  1.071804  bar  False  2012-01-01 00:00:00
```

Drop axis labels with missing data

drop ROWS if ALL column contains an empty value

```
>> df
        one       two     three four   five  timestamp
a       NaN -0.282863 -1.509059  bar   True        NaT
c       NaN  1.212112 -0.173215  bar  False        NaT
e  0.119209 -1.044236 -0.861849  bar   True 2012-01-01
f -2.104569 -0.494929  1.071804  bar  False 2012-01-01

>> df.dropna(how="a")ll
        one       two     three four   five  timestamp
a       NaN -0.282863 -1.509059  bar   True        NaT
c       NaN  1.212112 -0.173215  bar  False        NaT
e  0.119209 -1.044236 -0.861849  bar   True 2012-01-01
f -2.104569 -0.494929  1.071804  bar  False 2012-01-01
```