**Midterm -- Python**

Python Basics

* Basic Definitions
  + **Executive Bugs:** When the program is executing and discovers that it can’t legally carry out one of our instructions
  + **Intent bug:** program executes but doesn’t return the expected results
* Iterations and Functions
  + Example Functions
    - input() ⇒ asks for an input from the user, always gets it as a string
    - string\_name.lower() ⇒ lower case
    - string\_name.upper() ⇒ upper case
    - string\_name.split() ⇒ splits by spaces
  + While loop
    - While test\_expression:

spam=0

While spam<5:

print(“Hi”)

spam=spam+1

* “Body\_of\_while” can be a single statement or a set of statements
  + For loop
    - for i in range(10):

pres=input("Which U.S. president was born on July 4? " )

if pres=="Calvin Coolidge":

print("That is correct!")

break

else:

print("Nope, try again!")

* range(start, stop, step)
* Function: a set of statements that performs a specific task and can be executed by calling the name of the function
* Round a number in a fancy print statement to 2 decimals
  + Print(“I’ve got {**.:2f**} problems!”.format(variable))
* Creating functions
  + **Def** sample\_function(argument(s))
    - Body goes here
    - **Return** {variable goes here}
* Lists and Dictionaries
  + List functions (in format list\_name.append())
    - append():used to append an element at the very end of the list
    - insert(spot, what): used to insert at any location
    - remove(): remove a specific element from the list
    - sort(): sort the elements of a list
    - List\_name[::-1] = gets the list in reverse order
  + List Comprehension: a one-line way to create a list from another list
    - Example: compre\_list = [x+t for x in my\_list]
    - List functions
      * len(list), sum(list), min(list), max(list)
* Operations
  + % = Remainder/modulus
  + // = Integer division (round down)

NumPy

* Lists v. NumPy

Lists

* + Useful for heterogeneous lists, inefficient for homogenous data
  + Not contiguous memory allocation which would help time efficiency

NumPy

* + Efficiently store and operate on data arrays
  + Need to include “import numpy as np”
  + When you change a slice of NumPy the original array is altered too, unlike lists
* Create NumPy Arrays
  + From list: np.array(**[**list\_name, dtype=int**]**) *blue is optional to force a data type*
  + np.zeros(5) will create an array of 5 zeros
  + np.full(5, 3.14) will create an array of pi 5 times
  + np.arange(10,25,5) will create the array ([10, 15, 20])
* Slicing/Masking 2D NumPy arrays
  + array\_name[array\_name<7] returns all values less than 7
  + Ndarray slices are SEGMENTS not copies
    - When we use slice notation it returns a segment of the original. This means that if you change the elements of a slice, you will also change them in the original array
  + Let’s say we have array, called 2Darray:

1 2 3

4 5 6

7 8 9

* + 2Darray[0][2] = 3 (row 0, column 2 - remember implicit indexing)
  + 2Darray[ : , :1)=

1

4

7

* + 2Darray[:2][:1] = “in the first three rows, give me the first row”

1 2 3

* Othery NumPy
  + Array\_name.shape for dimensions, ex. (2,3) is a 2x3 grid
  + Also .dtype, .size, .itemsize (Which gives you the memory space it takes up)
  + You can import sys, then sys.getsizeof() to get the bytes your code uses
  + *%timeit -n 1* will evaluate how long one run takes
  + np.argsort() ⇒ returns the indices that would sort an array
* Universal Functions and Masking
  + Get new array of reciprocals of my\_array: 1/my\_array
  + Subtract each element from 10: 10 - my\_array
  + Np.{ insert Ufunc here } list
    - .argmin/.argmax (shows the index of the min/max values)
    - .all (are all true)
    - .any (are any true)
    - .median
    - .min/max
    - .var, .std (standard deviation), .mean
    - .add, .subtract, .multiply, .divide
    - .power (aka \*\*)
    - .abs (absolute value)
    - .unique (returns only unique values from array)
* Broadcasting: If any dimension the sizes disagree and neither is equal to 1, an error is
* NumPy Comparison Operators
  + == /// np.equal
  + != not equal /// np.not\_equal
  + < and <= /// np.less and np.less\_equal
  + > and >=
  + Examples *(Array Comparison)*
    - Which players aren’t 72 inches?
      * np.not\_equal(nd\_player\_heights, 72)
    - Which players over 200 lbs?
      * np.any(nd\_player\_weights > 200)
* Bitwise Boolean Operators (return T/F values)
  + & /// np.bitwise\_and (ex. data\_set
  + | /// np.bitwise\_or
  + ~ /// np.bitwise\_not (reverses an array’s significance; it’s not comparing)
  + Example
    - More array comparison:
      * (nd\_player\_heights >= 72) & (nd\_player\_heights <= 75)

Pandas

* DataFrame and Series: Loading, Selection and Operation
  + Loading: important pandas as pd
  + DataFrames are made up of an index and one or more Series
    - dataframe\_name.head() ⇒ gives the first few rows of the dataset
  + DataFrame Functionality
    - Create new sub dataset, first five rows of old: new = old [[‘age’,’gender’]][:5]
    - athletes\_data[‘Class’]==’FR’ will return a **boolean** list of all players with freshman labeled as True
      * To get list of freshman NAMES, do athletes\_data[athletes\_data[‘Class’]==’FR’]
    - athletes\_data[‘Height’].mean() will get mean of Height column
    - **Have to use iloc or loc when slicing/masking a dataframe**
      * Data.iloc[:1000] will get first 100 rows by implicit index
      * Data.loc [ :”name of index you want to stop at”] uses explicit
  + Series: the first column is the index, and the second is just a NumPy array
    - Can check if an index exists: “Notre Dame” in school\_series → True
  + Dictionary Like Features
    - Pandas maintains a mapping relationship between the index elements and the Series value, much like the key & values of a dictionary
  + Difference with the “in” operator in Series and DataFrame
    - Series = checks to is if a value is present in the lndex
    - DataFrame = checks to see if a value is present in the columns
* Handling Missing Data
  + Np.nan is a replacement for “None,” as we can actually interact with it
    - Can multiply and divide with this, but not sum/subtract
    - Use np.nansum() instead, it treats NaNs as 0
      * Also np.nanmean(), np.nanmedian()
    - NaN is a **float** datatype
  + UFuncs for missing values (doesnt change actual dataframe)
    - .isnull() - pulls up only missing data. Also .notnull()
    - .dropna(axis=’columns’) - drops any rows with missing values in them by default. Optional to clarify axis, can also do thresh=2 or how=”all” (only drops a row when **all** are NaN)
    - .fillna() - fills missing values with what you tell it to
    - Examples:
      * Number of missing values for each column: data\_set.isnull().sum()
      * Which column has most missing values? (2 lines of code)
        + Mark\_count\_null = mark\_data.isnull().sum()
        + mark\_count\_null.idxmax()
* Combining Data: Concat and Merge
  + Pd.concat is pandas equivalent to np.concatenate**(())**
  + Syntax: Pd.concat ([data1, data2], ignore\_index=True)
    - Can also add *, join=’inner’* or *, axis=1* (concat by columns)
  + Merging
    - Requires a common column or set of indices
    - pd.merge(df1, df2) when you have a common column and no complications
    - Add in **how** parameter to specify inner, outer, L, R join
      * Merge is inner by default
      * pd.merge(df1, df2, how=”outer”)
    - Combining 2 dataframes by common **column**
      * use left\_on and right\_on for a common column **that’s named differently**
      * Otherwise just do on=”name”
      * pd.merge(df1, df2, left\_on=”name”, right\_on=”athlete\_name”)
    - Combining 2 dataframes by common **index**
      * For differently named indices: Use right\_index=True or False, left\_index=True or False
      * Otherwise just do on=”name”
    - **Might have to use left\_on and right\_index in the same line of code**
    - Can also add in *suffixes=('\_x', '\_y')*
    - Should only have to use left\_on, right\_on, and how for a many to many merge
* Difference between a NumPy ndarray and a Pandas Series object is their indices
  + NumPy arrays have indexes as well, but they are implicit and always integers
  + Series do not have to integer based indexes (can be string, floats, etc.)
* Exporting
  + data.to\_csv(‘./filename/documentname.csv’)
* Importing
  + Var\_name = pd.read\_csv(‘./folder/documentname.csv’, index\_col=’Player’)
  + var\_name.head()

Homework and ICA Reading Questions and Answers

**Homework 1**

1. What is an *expression* and *statement* in a Python code? What is the distinction?
   1. A statement is a complete line of code that can have any purpose, whereas an expression can just be a portion of a line of code with values and operators inside it.

**Homework 2**

1. Blocks are isolated sections of code that are indicated by indentation. The three rules are:
   * 1. Blocks begin when the indentation increases
     2. Blocks can contain other blocks
     3. Blocks end when the indentation decreases to zero or to a containing block's indentation

**Homework 3 -** *These Answers Aren’t Necessarily True*

1. x1 = np.array([True, False])

x2 = np.array([False, True])

print(np.any(x1) and np.any(x2))

print(np.any(x1 & x2))

print(x1 & x2)

print(x1 and x2)

The third print statement yields the following output: [False False] You can visualize this code by imagining laying the two arrays on top of each other and checking if, in columns, both values are True. Since neither column lines up, the output will be an array that lists False twice.

The fourth print statement yields the following output: Error In regard to arrays, the "and" statement looks for two discrete, explicit true or false values - it doesn't know how to handle comparing multiple values inside an array, and returns an error.

1. Python lists can be used for any data type, and additionally do not have to remain completely heterogenous (each entry can be a different datatype) and be appended to easily without worry. However, the cost of these advantages are shortcomings in the data needed to run the code, and the speed at which the code can be run (lists require more of both). Additionally, lists cannot take advantage of Universal Functions, which save a lot of time in coding.

**Homework 4 - NumPy**

1. rev\_sort\_high\_2016 = sorted(high\_temps\_2016,reverse=True)
2. How many days had… >>>num\_days\_high\_2015 = np.**sum**(high\_temps\_2015<60)
3. boolean=precip\_list>.25

prec\_15\_16\_lo\_25 = precip\_list[boolean]

**Homework 5 -** *These Answers Aren’t Necessarily True*

1. In a pandas series, you can take advantage of indexes - this makes your data infinitely easier to call and manipulate, especially when in large quantities! However, it's important to note that this comes at the cost of speed; NumPy arrays operate much faster!
2. sample\_series.sort\_index(inplace=True)
3. Index preservation occurs when, after manipulating a dataframe with NumPy, the returned output mantains the pandas format and index that it had before
4. Index alignment: After executing an expression that utilizes multiple dataframes, pandas will know to align the indexes of the two dataframes *even if they aren't in the same order.*

**Homework 6 - Pandas**

1. All colleges in LA: college\_loan\_defaults[college\_loan\_defaults['city']=='LOS ANGELES']
2. Default rate greater than 40 or less than .5 = college\_loan\_defaults[(college\_loan\_defaults['year\_1\_default\_rate']>40) |(college\_loan\_defaults['year\_1\_default\_rate']<.5)]
3. 'UNIVERSITY OF NOTRE DAME' in colleges\_gt\_40\_lt\_5.index
4. Extract columns only: college\_loan\_defaults.loc[**:,** ["year\_1\_default\_rate","year\_2\_default\_rate","year\_3\_default\_rate"]]
5. perc\_female\_survived = len(titanic[titanic["Sex"]=="female"][titanic["Survived"]==1]) / (len(titanic[titanic["Sex"]=="female"])) \* 100
6. titanic.fillna(titanic.mean())

**ICA 1**

1. Keyword arguments are identified by the keyword put before them in the function call. Keyword arguments, frequently used for optional parameters, specify additional functionality within a function call.
2. **def** sum\_list(lis):  
    total=0  
    **for** elem **in** lis:  
    total=total**+**elem  
    **return** total  
   sum\_list([1,2,3,9])
3. The output should be: 5 3 Explanation: When we assign a list to a variable, changing the list will change the variable as well. However, in this instance, we assigned a *function* to list2 via list1, not a list. This means that the two variables are not referencing the same list, and therefore act independently.

**ICA 2**

1. df2=df1['StudentID','Score'] *new df with only two columns*
2. LeBron\_Tm = bball\_df['Tm']['LeBron James']
3. Kobe = bball[bball[‘Age’]>35][‘PTS\_per\_min’].argmax()