COMP1005 Week 8 Cheat Sheet

Lisa Luff

10/10/2020

Contents

Tuples **2**

Sets **2**
Set Operations 2

Dictionaries **3**

Pandas **4**

Data Structures **4**
Using Dataframes 5
Turning Dataframes into Arrays 6
Grouping Data in Dataframes 6
Slicing/Subsetting Dataframes 7

Reproducible Research **7**

Jupyter Notebooks **8**
Jupyter Notebook App 8
Running Jupyter 8

Tuples

- A tuple is anything within brackets, separated by commas
- Tuples are immutable, which means their structure cannot be changed
  - Length (So cannot append, or delete elements)
  - Ordered
- Can hold any type of information (not limited by type like an array)
- Can be created by:
  - *variable* = (‘*string*’, *float*, *integer*, etc.)
  - *variable* = *string*, *float*, *integer*, etc. (It will add brackets itself when stored)
- A tuple is a sequence so you can:
  - Use indexing
  - Use Vector operations

Sets

- Set are defined with {}
  - *variable* = {‘a’, ‘e’, ‘i’, ‘o’, ‘u’}
- Objects in a set are the *elements* of the set
- Sets are not a sequence
  - Sets are unordered
  - There are no duplicates
    - {1, 2, 3} = {2, 3, 1} = {2, 3, 3, 1, 2}
- Allows you to use in and not in for if statements
  - if *thing* in *set*:
- Can be written in predicate form:
  - {1, 2, 3, 4} = {X:X is a positive integer less than 5}
    - Where the : represents | (or such that)
- An empty set {} = 0

Set Operations

- Sets are essentially the same sets used in statistics
- **In python:**

- Set creation
  - *set1list* = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’] → *set1set* = **set**(*set1list*)
  - *set2list* = [‘g’, ‘c’, ‘h’] → *set2set* = **set**(*set2list*)

- Intersection
  - intersection = *set1set*.**intersection**(*set2set*) → intersection = {‘c’}

- Union
  - union = *set1set*.**union**(*set2set*) → union = {‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’, ‘h’}

- Differences
  - difference*12* = *set1set*.difference(*set2set*) → difference*12* = {‘a’, ‘b’, ‘d’, ‘e’, ‘f’}
  - difference*21* = *set2set*.difference(*set1set*) → difference*21* = {‘g’, ‘h’}

Dictionaries

- Dictionaries map keys to values
  - In a traditional dictionary, words are "mapped" to the meaning of that word
- Our code might want to map
  - Town → population
  - Month → total rainfall
  - Student ID → student name
- The dictionary holding these keys is as set of key:value pairs
  - We can add, delete and overwrite dictionaries
  - Keys themselves must be immutable
- In use the dictionary is kind of like a class, with each key:value pair an instance
- **In python:**

- Creating the dictionary
  - *dictionary* = {'Me' : 'Lisa', 'Love' : 'Wynand', 'Baby' : 'Cat'}

- Adding to the dictionary (don't need to append or anything)
  - *dictionary*['Mom and Dad'] = 'Michele and Ian'

- Deleting an entry
  - del *dictionary*['Mom and Dad']

- Accessing the keys alone using keys()
  - *dictionary*.keys() = dict_keys(['Me', 'Love', 'Baby'])

- Using keys to access their values
  - print(*dictionary*['Me']) → output - Lisa

- Accessing the values specifically using values()
  - plt.bar(*dictionary*.keys(), *dictionary*.values())
- Just the keys
  - for item in *dictionary*: → print(item) →
    Me
    Love
    Baby

- Just the values of the keys
  - for k in *dictionary*.keys() → print(*dictionary*[k]) →
    Lisa
    Wynand
    Cat

- Combining keys and values with keys()
  - for k in *dictionary*.keys(): → print(k, ':', *dictionary*[k]) →
    Me: Lisa
    Love: Wynand
    Baby: Cat

- Counting values
  - Using random to "toss a dice" and counting how many times each value is rolled
    - dicecount = {1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6:0}
    - for i in range(1000): → rollvalue = random.randint(1, 6) → dicecount[rollvalue] += 1
      - So essentially this adds one to the value assigned to the key associated with the value that was rolled

- Finding frequencies
  - Counting the frequencies of words in a book
    - words = [*words in book separated by commas*]
    - wordfreq = {} (empty set)
    - for word in words: → if word not in wordfreq: →→ wordfreq[word] = 0 → wordfreq[word] += 1
      - In this case it adds new words as a key:value pair of word:0, and then whether the word is new or not, it adds 1 to value associated with the word

- Lots of packages can be used with dictionaries

Pandas

- Use pandas to create data frames
- The pandas library
  - Provides data structures (dataframes)
  - Works well with matplotlib and NumPy
- Using pandas
  - import pandas as pd

Data Structures

- Data structures = dataframes
- Dataframes
  - 2 dimensional
  - Data stored in columns
  - Data can be any type
- Uses standard indexing (0-based)
  - Index refers to the position of an element in the dataframe
  - Index values can be overridden, but causes issues (don't)
- Dataframes are essentially dictionaries
  - Keys are column labels
  - Values are a list of values

- **In python: (after importing pandas)**

- Creating the dataframe using DataFrame()
  - *df* = pd.DataFrame({'*Label1*': [1, 2, 3], '*Label2*': [4, 5, 6], '*Label3*': [7, 8, 9]}) →

Index	*'Label1'*	*'Label2'*	*'Label3'*
0	1	4	7
1	2	5	8
2	3	6	9

Using Dataframes

- Gathering basic info for numeric data with describe()
  - *df*.describe() →

Info	*'Label1'*	*'Label2'*	*'Label3'*
count	3	3	3
mean	2	5	8
std	1	1	1
min	1	4	7
25%	1.5	4.5	7.5
50%	2	5	8
75%	2.5	5.5	8.5
max	3	6	9

- Reading in a CSV file with pandas into a dataframe
  - **No need for splitting, etc. All done for you!!!**
  - *csvdf* = pd.readcsv("*filename.csv*")
    - Automatically assigns column names and everything!
- Types within dataframes
  - type(*df*) - Will tell you that it is a class defined as a pandas dataframe
  - *df*.dtypes - Will give a list of the keys with the type of the values stored in that key
    - Object is usually strings
- Describing the dataframe itself
  - *df*.columns → Index([*listofkeynames*], dtype='object')
  - *df*.shape → (*numberofvaluestotal*, *numberofkeys*)
  - *df*.head() → table with keys, index 0 to 4 and value of the element at that index for each key
    - tail() - Does the last 5 elements of each
    - You can specify how many elements to show for head and tail by putting the required number in the brackets

Turning Dataframes into Arrays

- You can turn the dataframe into a list of arrays, where each array is assigned to the key and contains the column of data
  - *df*.columns.values
    - → array([‘*keys*’], dtype = object)
    - Then you can access the arrays
      - *df*[‘*columnname*’]
- Finding unique values within a column, creates a list of arrays with unique values each assigned to an array with information about the instances that value occurs in
  - pd.unique(*df*[‘*columnname*’])
    - → array([‘*uniquevalues*’], dtype = object)
- You can use describe() for specific columns
  - *df*[‘*columnname*’].describe()
- Functions for gathering specific data
  - .min()
  - .max()
  - .mean()
  - .std()
  - .count()

Grouping Data in Dataframes

- You can group/sort data with groupby()
  - *sorteddf* = *df*.groupby(‘*columnname*’)
- Then you can run statistics on the sorted data
  - *sorteddf.statfunction*()
- We can count how many instances of each unique entry there are for a column
  - *typecount* = *df*.groupby(‘*columnname*’)[‘*alluniquecolumn*’].count()
    - This essentially counts how many values from the other column there are for each unique value in the column of interest. The other column needs to be a unique identifier though (id number, etc.)
- Or we can count only the number of instances for a specific unique value in a column
  - *df*.groupby(‘*columnname*’)[‘*alluniquecolumn*’].count()[‘*valueofinterest*’]
- We can do maths on the arrays the same as any other array
  - *df*[‘*columnname*’]+-*/*n*
- Plotting can be really easy
  - *typecount*.plot(kind = ‘*plottype*’)

Slicing/Subsetting Dataframes

- Slicing is done exactly the same, but when you slice a dataframe, you are selecting rows (or elements of the array associated with the specified column)
  - Don't forget the last value specified is not included when slicing
- iloc indexes by position
  - *df*.iloc[*rowslicing*, *columnslicing*] **still exists**
    - eg, *df*.iloc[0:2, 2:7] for a range OR
    - *df*.iloc[[*listofrowvalues*], :] for specific rows
- **Current system**
  - Can use iloc when indexing rows and columns
  - For specifying rows for all columns
    - *df*[*rowindexing*] (don't need loc or iloc) for a range
  - For specifying all rows for specific column labels
    - *df*[[*listofcolumnnames*]] (don't need loc or iloc)
  - You cannot combine row indexing with calling specific column labels and vice versa
    - Instead create a subset with the list of labels using the above methods, and then use the above methods to index that
- #loc indexes by labels **no longer exists**
  - #*df*.loc[*rowslicing*, *listofcolumnnames*] OR
  - #*df*.loc[*listofrows*, *columnslicing*] OR
  - #*df*.loc[*listofrows*, *listofcolumns*]
    - #Everything within a specified list is included (does not exclude last value as it is not a range)
    - #eg, *df*.loc[:, [*column1*, *column2*]]
- You can use boolean operations to create subsets
  - *df*[*df.columnname == valueincolumn*]
    - This will create a data set with all columns of information for instances that have that set value in the specified column
  - You can use !=, >=, <=, etc.
- **Note**
  - Using = just references (or points to an existing object) whether it be an array, list, dataframe, etc.
  - If you want to create a true copy (create a completely new, separate object the exact same as the existing object), use the .copy() function
    - *copyvariable* = *df*.copy()
  - **This is important** because even if a slice or subset exists under a separate variable name, it just points to that section of the original
    - This means any changes to the data in the variable holding the slice/subset, will **make those changes in the original!**

Reproducible Research

- We should be able to reproduce any research
- As such journals now ask for the data and code used with the submitted papers to check and verify the results
- We can use various programs that allow insertion and running of blocks of code within a written report
- This can be combined in a notebook
- Notebooks let you
  - Present workflows
  - Show overall logic
  - Refine analysis/workflow over time
  - Create presentations

Jupyter Notebooks

- Born out of IPython Project in 2014
- Aims to support interactive data science and scientific computing across all programming languages
- 100% open-source
- Free for all use, released under the liberal terms of the modified BSD license
- Used by everyday users through to top of the top scientists and developers
- Contain both computer code, and rich text elements (paragraph, equations, figures, links, etc.)
- They are both
  - Human readable
  - Executable documents that can be run to perform data analysis

Jupyter Notebook App

- The server-client application that allows displaying, editing and running of notebook documents via a web browser
  - Can be run on a local desktop with no internet connection, or installed on a remote server and accessed through the internet
- There is also a dashboard/control panel
  - Shows local files and allows you to open notebook documents, or to shutdown their kernals
  - It also acts similarly to a file manager, allowing you to navigate, rename and delete files
- Notebook kernals
  - A kernal is the computational engine that executes the code in a notebook document
  - The ipython kernel executes python code
    - You can chose what version of python the kernal runs
  - Kernals for many languages
  - When you open a notebook document, the associated kernal is automatically launched

Running Jupyter

- In terminal, type jupyter notebook in the directory with your notebooks
- To shut down jupyter, type control-c in the terminal window you ran jupyter from
- In the dashboard - click new (top right corner) to create a new notebook
- Use drop down menu to chose what to enter
  - Code
  - Markdown
  - Raw NB convert
  - Heading
- shift-enter runs inserted code
- Click on "Untitled" to rename the file
  - File will be *filename*.ipynb
- You can cut, paste, edit and re-run cells of code
  - Changes to earlier cells can impact later ones if they're re-run
- Type %run *script*.py to run python programs in the notebook if in the same directory
  - Or %run *path*/*script*.py
- You can save a matplotlib.pyplot graph created in a notebook with plt.savefig('*imagename.filetype*) after plt.show()
- Different co-currently running notebooks will be in different browser tabs
- Notebooks also maintain metadata and context information
  - %reload_ext version_information
  - %version_information numpy, scipy, matplotlib
- Notebooks can also be shared