CS 320 Learning Objectives

Reproducibility:

- define reproducibility in the context of data science
- write code that is reproducible across different operating systems
- create, configure, and access a virtual machine
- identify the key responsibilities of an interpreter

Git/Versioning:

- switch between snapshots (commits) in a repository
- complete the steps of creating new snapshots (commits) of code
- identify use cases for tagging, branching, and merging
- manually resolve a merge conflict
- push/pull commits to/from a remote repository
- create a pull/merge request to a repository

Benchmarking and Automation:

- write code to automate the execution of programs
- capture the output of a program and decode it to a string
- handle exceptions caused by programs that crash
- measure how long a program or piece of code takes to run
- visually relate input size to runtime

Complexity Analysis:

- identify steps in a piece of code
- count how many steps will execute given a specific input
- find a function to relate input size to execution count
- use complexity classes based on multiple variables
- classify functions/algorithms by complexity class
- recall complexity of common data structure operations
- optimize inefficient code to produce a better complexity class
- remember the following complexity classes: O(1), O(log N), O(N), O(N log N), O(N^2), O(N^3), and O(N!)

Large Data:

- write code to read and write compressed data
- write code to enumerate contents of compressed archives and compute compression ratio
- use yield statement to create generator functions
- optimize memory usage of inefficient code by replacing the population of large data structures with generators
- identify limitations of generators related to random access

- trace through execution of a loop over a generator

OOP:

- refactor code using dictionaries to represent entities to code that represents them with objects instead
- trace through method calls
- create classes that inherit from other classes (single inheritance only)
- write methods that override other methods
- call overridden methods
- identify which method will be called given an inheritance hierarchy (single inheritance only)
- inspect method resolution order (__mro__) to understand the implications of inheritance on what method will be dispatched, including for hierarchies involving multiple inheritance
- use special methods, including constructors, correctly
- create and use context managers
- create and use subscriptable objects
- create and sort sortable objects
- identify cases where context managers can help with guaranteeing cleanup
- decompose bigger problems into smaller classes or functions

Recursion:

- identify recursive patterns in mathematical expressions
- identify a base case
- convert a recursive mathematical expression into code
- trace recursive functions that return something
- trace recursive functions that do something
- write recursive code, including cases involving traversal of nested structures

Graphs:

- map real-world problems into terms of nodes/edges
- visualize graphs with graphviz
- define important graph terms, including node, edge, metadata, path, connected, weakly connected, cycle, directed graph, parent, child, DAG, tree, root, leaf, and binary tree
- answer questions about what properties a given graph has
- implement graphs with objects
- write recursive code to visit all (or select) nodes of a tree

BSTs

- write code search a regular binary tree for values
- define the BST rule and determine whether a given tree follows it (left descendents < mid < right descendents)
- write code to efficiently search a BST for a value
- recursively construct a BST (in code, on paper)

- know what balance means for performance and implication of insert order
- compute a tree's height recursively

DFS:

- understand why we often need a separate graph class (nodes dict)
- implement basic DFS algorithm without support for cycles
- implement visited set to support cycles
- use DFS to discover all reachable nodes
- use DFS to determine if there is a path
- use DFS to identify the specific path, if it exists

BFS:

- implement BFS using a task queue to keep track of future nodes to visit
- develop a strategy to prevent BFS from repeatedly visiting the same nodes
- use BFS to discover all reachable nodes
- use BFS to determine if there is a path
- use BFS to identify the specific path, if it exists

Queue structures

- understand the behavior of stacks, queues, and priority queues
- identify and use efficient Python structures (lists, deques, and heapq) for each of the above

Search (general):

- identify when BST, DFS, BFS can/should be used?
- use DFS/BFS in contexts where nodes don't correspond to in memory structures and graphs may be of infinite size

Web Scraping:

- understand the role of JavaScript has in loading page content on many pages
- use both requests module and selenium to scrape pages
- determine whether requests or selenium is more appropriate for a specific page
- automate clicks and typing with selenium
- write polling code that waits for something to complete
- use Selenium to search for elements in the DOM
- combine web scraping with graph search to crawl multiple pages
- scrape respectfully by considering robots.txt files and 429 responses

Dashboards:

- write event-driven web handlers to respond to URL requests
- create dashboards consisting based multi-resource web pages
- generate figures "on-the-fly" in a web application based on updating data
- create handlers that return binary data, with content type correctly identified in response headers

- write handlers that take POST data and query strings as input

A/B testing:

- recall reasons one might want to A/B test
- identify useful metrics to compare for A/B testing
- identify possible treatment factors
- generate a contingency table corresponding to an A/B test
- compute and interpret a p-value for a contingency table
- identify the tradeoffs between user A/B testing and request A/B testing
- recall three ways to split users into A/B groups (IP, sign ins, cookies) and the tradeoffs between each approach

Regular expressions:

- recall regular expression language for character classes, meta characters, repetition, and anchoring
- use regular expressions in code for search and replacement
- write regular expressions that decompose matches into meaningful groups
- determine whether a given pattern will match a given string

Custom Plots:

- differentiate between visual coordinates and data coordinates
- identify the correct coordinate system to use for a given drawing need
- identify the key coordinate systems in use for a matplotlib figure with subplots
- annotate plots with lines, arrows, rectangles, circles, and text
- draw objects where one aspect (e.g., position) is based on one coordinate system and another aspect (e.g., size) is based on a different coordinate system
- configure DPI to create higher/lower quality plots
- control z order of shapes
- control transparency of shapes

Geo Data:

- read and write shapefiles and geojson using geopandas
- understand the implications of coordinate reference systems for measuring distance and area
- write code to re-project between different coordinate reference systems
- use pre-available methods for computing intersection, union, and difference of geometric areas
- use geocoding services to translate addresses to coordinates

Matrices and Linear Algebra:

- compute dot product
- use existing modules to solve linear matrix equations
- represent graphs as matrices, where each row is an edge, or where every cell is a connection between two nodes

- represent raster data as matrices (land use, road quality)
- represent images as 3D tensors

Machine learning (general):

- clean data based on documentation and exploratory analysis
- understand how to correctly fit and predict with respect to a train/test split
- identify the correct kind of ML problem (regression, classification, clustering, decomposition) given a plain-language description of the goal
- correctly use train and test datasets to fit and score a model
- use cross validation scoring to produce stable performance metrics
- describe gradient descent
- tune iterative algorithms (such as LogisticRegression)
- write code for iterative algorithms (such as K-Means)
- write custom pipeline code that produces transformed output based on the inputs, without mutating state

Features:

- identify cases where the following feature transformations are needed: polynomial features, standard scaling, one-hot encoding
- write code to build ML pipelines with both transformers (in parallel or in series) and models

Regression

- perform least-squares regression via matrix operations
- use existing modules to perform least-squares regression
- interpret "explained variance" scores for a regression model
- visualize coefficients for multiple regression models
- visualize prediction lines for single regression models

Classification

- use existing modules to perform LogisticRegression classification, making both categorical and probabilistic predictions
- calculate and interpret confusion matrices, accuracy, precision, recall, false positive rate, and false negative rate

Clustering

- apply existing modules to cluster using k-means and AgglomerativeClustering
- describe process KMeans uses to find clusters, including how n_init and n_clusters work
- write code to visualize hierarchical clusters (dendrograms)
- identify a "good" number of clusters for a k-means clustering

Decomposition

- use existing modules to perform singular value decompositions (SVD) on matrices
- apply SVD to perform lossy compression on data tables

- use existing modules to perform PCA analysis
- interpret explained variance scores of PCA to determine appropriate number of components to use

Parallelism

- write matrix code to run on GPUs
- understand the performance tradeoffs between GPUs and CPUs
- differentiate between threads and processes
- write code that shards computation over multiple processes