Recall: Data analysis

- access data
- file processing
- inspect (clean) data
- store data in Python objects/data structures

- analyze data, e.g.,
  - mean
  - range
  - median
  - standard deviation
  - mode

- report on results of data analysis, e.g.,
  - frequency occurrence table
  - frequency occurrence chart

Data mining:

Application of automated techniques that attempt to discover underlying patterns in the data.

For example, cluster analysis:

Data mining technique that attempts to divide the data into meaningful groups called clusters.

A primary goal of data visualization is to communicate information clearly and efficiently via statistical graphics, plots, and information graphics.

Numerical data may be encoded ... to visually communicate a quantitative message.

Effective visualization helps users analyze and reason about data and evidence. It makes complex data more accessible, understandable and usable.
def visualizeQuakes(k, r, fname):
    """ (int, int, str) \rightarrow None
    
    Data mining.
    """
    qdatadict = readFile(fname)  # access file data
    qCentroids = createCentroids(...)  # process data
    qClusters = createClusters(...)  # cluster analysis
    eqDraw(...)  # report - visualization
    return None

Data mining:
application of automated techniques that attempt to
\textit{discover underlying patterns} in the data.

Cluster analysis:
data mining technique that attempts to \textit{divide the data}
\textit{into meaningful groups} called clusters

Cluster:
data values that show some kind of similarity to each
other while exhibiting dissimilarity to data values
outside the cluster

IMPLEMENTING K-MEANS CLUSTER ANALYSIS
1) review k-means clustering algorithm
2) work some examples – simple data, earthquake data
3) review high level structure of program – key functions
4) review important data structures (including earthquake data file)
5) code from the bottom up
6) testing each function thoroughly before proceeding to the next
7) integrated testing at each level
1) review k-means clustering algorithm
   -- decide how many clusters (k)
   -- assign each item from the data set to a cluster

Clustering the data – k-means clustering algorithm
1) decide how many clusters (k)

Choose k depending on the data and desired results.
For example:
   too hot, too cold, just right
   group1, group2, group3, ...., group 10

Run the analysis for different values of k.

k-means cluster analysis
1) decide how many clusters (k)
2) assign each item from the data set to a cluster:
   determine similarity of the item to other data in the cluster
   assign item to cluster where data are most similar

3) do this until done

Clustering the data – k-means clustering algorithm
2) assign each data item to a cluster:
   by measuring the distance from a data item to the centroid of
   each cluster, then choosing the closest cluster
   a) need initial centroids for initial clusters to form around
   b) need to be precise about similarity (distance) measure
   c) need to determine when to stop
**Initial centroids?**

→ choose k random points from the data set for starter centroids.

**Precise about similarity/distance measure?**

Earthquake data items are points
→ use distance between two data points:
  -- data point from the data set
  -- a cluster centroid point

**Distance: Euclidean distance**

\[ d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \]

**Each cluster is a collection of data points**

associated - measured by Euclidean distance - with the centroid of the cluster.

**Centroid:**

mean of a cluster (collection of data points)

*Note that, after the initial set of centroids, the centroid might not be a member of the collection of data points.*

**k-means cluster analysis**

1) decide how many clusters (k)
2) assign each data item to a cluster:
   randomly choose k of the data points to serve as starter centroids for the k clusters.
   assign each data point to the cluster with the centroid that is the closest (Euclidean distance) to the data point
3) do until done:
   c) need to determine when to stop

**K-Means Cluster Analysis Algorithm**

3) repeat until done: need to determine when to stop

recalibrate the centroids of each cluster
re-assign data items to clusters

**e.g., # repetitions, clusters are stable**

this algorithm: choose a reasonable value for r
### $k$-means cluster analysis

1) decide how many clusters ($k$)  
2) assign each data item to a cluster:  
   - randomly choose $k$ of the data points to serve as starter centroids for the $k$ clusters.  
   - assign each data point to the cluster with the centroid that is the closest (Euclidean distance) to the data point.  
3) until done (clusters created $r$ times):  
   - recalibrate the centroid (mean) for each cluster  
   - assign each data point to the cluster with the centroid that is the closest (Euclidean distance) to the data point.

---

```python
def visualizeQuakes(k, r, fname):
    ''' (int, int, str) \rightarrow None
    Data mining.
    >>> visualizeQuakes(6, 7, 'earthquakes.txt')
    '''
    qdatadict = readFile(fname)  # access file data
    qCentroids = createCentroids(...)  # process data -
    qClusters = createClusters(...)  # cluster analysis
    eqDraw(...)  # report - visualization
    return None
```

---

For Example:  
```
data points = [34, 44, 10, 99, 50, 67, 0, 29, 50, 100]
k - ? design decision, e.g., 3 (low, medium, high)
centroids = [10, 50, 67] (initially, random choices from data)
clusters = [[], [], []] (first round)
clusters = [[[0], [10, 29]], [34, 44, 50, 50], [67, 99, 100]] (second round - use new centroids to create new clusters)
clusters = [[], [], []] (mean of each second round cluster)
clusters = [[0, 10], [29, 34, 44, 50, 50], [67, 99, 100]] (repeat until done)
```

---

For Example:  
```
data points = [34, 44, 10, 99, 50, 67, 0, 29, 50, 100]
k - ? design decision, e.g., 3 (low, medium, high)
centroids = [10, 50, 67] (initially, random choices from data)
clusters = [[], [], []]  
centroids = [10, 50, 67, 67, 99, 100] (mean of each cluster)
clusters = [[0], [10], [29, 34, 44, 50, 50]]
```

---

For Example:  
```
data points = [34, 44, 10, 99, 50, 67, 0, 29, 50, 100]
k - ? design decision, e.g., 3 (low, medium, high)
centroids = [10, 50, 67] (initially, random choices from data)
clusters = [[], [], []]
clusters = [[0, 10], [29, 34, 44, 50, 50], [67, 99, 100]]
```

---

For Example: using a data dictionary (createClusters)  
```
data points = {1: [34], 2: [44], 3: [10], 4: [99], 5: [50], 6: [67], 7: [0], 8: [29], 9: [50], 10: [100]}
k - ? design decision, e.g., 3 (low, medium, high)
centroids = [10, 50, 67] (initially, random choices from data)
clusters = [[], [], []]
clusters = [[0], [10, 29], [34, 44, 50, 50], [67, 99, 100]] (mean of each cluster)
clusters = [[], [0, 10], [29, 34, 44, 50, 50], [67, 99, 100]] (mean of each new cluster)
clusters = [[0, 10], [29, 34, 44, 50, 50], [67, 99, 100]] (mean of each new cluster)
```
For Example: using a data dictionary

data dict = {1: [144.8897, 38.0555], 2: [45.5993, 34.9144], 3: [45.9411, 34.8857], 4: [58.2176, 9.1259], 5: [168.5847, -21.5194], 6: [-14.0995, -11.7482], 7: [168.4966, -21.4067], 8: [73.9098, 38.2556], 9: [129.9681, -7.0579], 10: [-84.0888, -41.4839], 11: [168.5912, -21.5266], 12: [141.5594, 32.4901]}

centroids = [[-84.0888, -41.4839], [141.5594, 32.4901], [144.8897, 38.0555], [45.5993, 34.9144], [73.9098, 38.2556], [168.5847, -21.5194]]

# clusters are lists of dictionary keys
[[5, 7, 11], [2], [3, 4, 8], [6], [1, 9, 12], [10]]

IMPLEMENTING K-MEANS CLUSTERING

✓ review k-means clustering algorithm
✓ work some examples – simple data, earthquake data

1) review high level structure of program – key functions
2) review important data structures (including earthquake data file)
3) code from the bottom up
4) testing each function thoroughly before proceeding to the next
5) integrated testing at each level

CIS 210

def readFile(filename):
    with open(filename, "r") as datafile:
        datadict = {}
        key = 0
        for aline in datafile:
            items = aline.split()
            key = key + 1
            lat = float(items[3])
            lon = float(items[4])
            datadict[key] = [lon, lat]
        return datadict

CIS 210
def createCentroids(k, datadict):
    centroids=[]
    centroidCount = 0
    centroidKeys = []
    while centroidCount < k:
        rkey = random.randint(1,len(datadict))
        if rkey not in centroidKeys:
            centroids.append(datadict[rkey])
            centroidKeys.append(rkey)
            centroidCount = centroidCount + 1
    return centroids

CIS 210
def createClusters(k, centroids, datadict, r):
    on each pass (for a total of r passes):
    (1) set clusters to empty, e.g., [[], [], []]
    (2) for each key in datadict
        for each centroid
            compute distance between datadict value(s) and the centroid
        choose the minimum distance
        put key in corresponding cluster
createClusters\(k, \text{centroids, datadict, r}\), cont’d on each pass (until r passes):

(3) for each cluster
determine the mean of the cluster
update centroids list with that mean

[repeat – next pass – until r passes]

eqDraw

note that len(colorlist) must be \(\geq k\)

IMPLEMENTING K-MEANS CLUSTERING

✓ review k-means clustering algorithm
✓ work some examples – simple data, earthquake data
✓ review high level structure of program – key functions
✓ review important data structures (including earthquake data file, data dictionary, centroids, and clusters)
✓ code from the bottom up
1) testing each function thoroughly before proceeding to the next
2) integrated testing at each level

IMPLEMENTING K-MEANS CLUSTER ANALYSIS

1) testing each function thoroughly before proceeding to the next
2) integrated testing at each level

A word on integrated testing …
--- start with very small data sets
--- run program for different values of k and r
--- run program for different data sets

Data Analysis \(\rightarrow\) Data Mining
• Recursion

à experimenting
Define a function `stars` such that:

```python
>>> stars(1)
*
>>> stars(2)
* ** *
>>> stars(3)
* *** * ** *
>>> stars(4)
* *** * ** * **** * ** * *** * ** *
```

Do you see a pattern?

stars(1) *
stars(2) * ***
Do you see a pattern?

stars(1) *
stars(2) * ***
stars(3) * *** ** ***
stars(4) * *** ** *** **** ** **
Do you see a pattern?

stars(1) *
stars(2) * ***
stars(3) * *** ** ***
stars(4) * *** ** *** **** ** **
stars(5) ...
Do you see a pattern?

Each step of the problem can be described in terms of the prior step, i.e., a simpler version of the problem. \( \rightarrow \) **Recursive solution**
Do you see a pattern?

What is (are) the most simple version(s) of the problem?
Or, What is (are) the base case(s)?

What should happen when it is not a base case?
Or, What is the recursive rule?
Or, How can we break the problem into a simpler version of itself?

Each step of the problem can be described in terms of the prior step, i.e., a simpler version of the problem. → Recursive solution

What is the base case? → ??
What is the recursive rule? → ??

```python
def stars(n):
    '''
    if n == 1:
        print('*')
    else:
        stars(n-1)
        print(n * ' ')
        stars(n-1)
    return None
```

```python
def stars(n):
    '''
    if n == 1:
        print('**', end=' ')
    else:
        stars(n-1)
        print(n * '**', end=' ')
        stars(n-1)
    return None
```
Recursion

- A powerful approach to problem solving approach, where a problem is solved by reducing it to a simpler version of the original problem (and combining those solutions).
- A function that calls itself (to solve the simpler version of the original problem).

Recursive function:

def countdown(n):
    '''
    if n == 0:
        # base case
        print('Blastoff!')
    else:
        print(n)
        countdown(n-1)  # recursive call
    return None

>>> countdown(4)

def countdown(n):
    ''' For loop is better for this problem '''
    for i in range(n, 0, -1):
        print(i)
    print('Blastoff!')
    return None

def stars(n):
    ''' Problem has recurring pattern \rightarrow recursive solution '''
    if n == 1:
        # Base case
        return('*')
    else:
        # Recursive rule
        return stars(n-1) + (n * '*') + stars(n-1)

CIS 210

Python/programming toolkit so far

- Assignment statement
- Expressions
- Python repetition -- for, while
- Python conditionals -- if
- Numeric data types (int, float) and operations (e.g., +, *, round, abs)
- Boolean data type and operations (e.g., <, and)
- String data type and operations (e.g., +, len, count, find, format), formatted strings
- Python collections data types and operations -- tuples, lists, dictionaries
- Python files and file processing
- Data type "coercion" functions (e.g., str, int, list, float)
- None/Type (None)
- Print/input
- Python Standard Library -- math, turtle, random modules; import (if __name__ == '__main__')
- User-defined functions; function design; docstrings
- IDLE interactive development environment
- Python introspection: help, dir, type, id
- Run-time checking of data and code: assert; try/except
### CIS 210 Learning Outcomes

- understand, develop, implement algorithms for computational problem solving;
- use structured design and testing methods to develop and implement programs;
- read, write, revise, document, test, and debug code;
- demonstrate robust mental models of data representation and code execution;
- demonstrate good understanding of a high level programming language;
- introduce and/or implement a sampling of classic computer science problem domains and algorithms.