CSC108H Lecture 7

Dan Zingaro
OISE/UT

June 25, 2009

Clickers: Dictionary Review

Assume we initialize a dictionary `d` like this:

```
d = {1:2, 3:4}
```

Which of the following **does not** return the value associated with key 3?

- A. `d[3]`
- B. `d.get(3)`
- C. `d.keys()[3]`
- D. `d.get(3, -1)`

Dictionaries as Flexible Lists

- When using lists, it is illegal to access or assign to an index that is off the end of the list
- (We'd have to preallocate a list such as `[0] * 100`)
- A dictionary that uses integer keys "looks like" a list that seems to grow when we assign to a new index
- For example, if we assign `d[99] = 'blah'`, we can retrieve it with `d[99]` without wasting any memory on 99 unused entries

Dictionaries as Records

- Other languages have a record (Pascal), struct (C/C++), or similar data type to represent records consisting of multiple fields (e.g. for a person: name, age, interests)
- We can use Python dictionaries for this: the keys represent the field names, and the values represent the values of the fields
- Question: what's wrong with using a list of three elements (the third of which would be a nested list) instead?

```
dan = {
  'name':'dan', 'age':26,
  'interests':['teaching', 'icecream']}
```

Dictionaries as Records (record.py)

- Imagine we have a dictionary where each key is a person's name and each value is the dictionary of their data ...

```
people = {
  'dan':{
    'name':'dan', 'age':26,
    'interests':['teaching', 'icecream']},
  'joe':{
    'name':'joe', 'age':22,
    'interests':['video games', 'biking', 'sleeping']},
  'steph':{
    'name':'steph', 'age':24,
    'interests':['biking']}}
```

Dictionaries as Records...

- Let's write a function that collects the interests of each person from such a dictionary into a list
- The list should not contain duplicates
- Plan: for each dictionary in the main dictionary, collect the interests and add them to a list (as long as they are not already there)
- ...

Object Persistence

- Think of our `people` dictionary as a database that we might add to or modify
- But, problem: when we close Python, our "database" is gone!
- We could come up with a file format for storing our database, and write a function that writes a dictionary in this format

```
dan
26
teaching|icecream
end
joe
22
video games|biking|sleeping
end
steph
24
biking
end
```

Pickle Module

- What if a user gives `end` as one of their interests? What happens if we later add a field to our records?
- It is much simpler to use the Python pickle module to implement persistence
- If we `import pickle`, we can use `pickle.dump (obj, file_obj)` to write a representation of `obj` to `file_obj`
  - `file_obj` must be a file open for writing
- We can use `pickle.load(file_obj)` to later get the dumped object back out of our file
  - `file_obj` must be a file open for reading

Sparse Matrices

- A sparse matrix is a matrix whose entries are almost all zero

$$\begin{pmatrix} 0 & 0 & 4 \\ 0 & 0 & 0 \\ 0 & 3 & 0 \end{pmatrix}$$

- Storing sparse matrices as lists of lists can waste a lot of memory
- Alternative: use a dictionary whose keys are (row, column) tuples and whose values are the values at those coordinates
- e.g. for the above: `{(0, 2):4, (2,1):3}`

Sparse Matrices...

- To add two matrices, we add their corresponding components

$$\begin{pmatrix} 0 & 0 & 4 \\ 0 & 0 & 0 \\ 0 & 3 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 1 \\ 0 & 12 & 0 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 5 \\ 0 & 12 & 0 \\ 0 & 3 & 0 \end{pmatrix}$$

- Task: write a function that takes two sparse matrices stored as dictionaries and returns a new dictionary representing their sum

Artist Filtering

- Some people like to listen to different types of music depending on what they're doing
- For example, we might listen to soft music when working in the garden, something with memorable words if we feel like singing, and something a little more upbeat when driving
- We may even have our singing-in-the-garden music that is different from just-gardening music
- The function we'll write will take information about the artists we like to listen to in different circumstances (reading, cleaning, singing, dancing, and so on), and prune the list based on inclusion/exclusion criteria

Artist Filtering...

- Our function will use a dictionary of artist data that will be read in from a text file
- The file stores each "artist block" as the artist's name, the information for that artist, and a line containing only "end"

```
coldplay
driving
cleaning
walking
gardening
end
usher
sleeping
end
```

Artist Filtering...

- We start with a list consisting of all available artists, and a dictionary mapping each activity to a list of artists
- Given an activity and a list of artists, there are two ways we will want to be able to filter
  - Inclusion: include **only** those artists that we want to listen to during the activity
  - Exclusion: include **only** those artists that we **do not** want to listen to during the activity
- Let's go through some examples with `artists.txt`
- Which artists remain after **including** gardening and then (on this new subset) **including** singing?
- Which artists remain after **excluding** gardening and then **including** singing?
- Which artists remain after **excluding** gardening and then **excluding** thinking?

Artist Filtering...

```
def artist_filter(artist_dict, artist_list,
                  activity, include):
```

- The `artist_filter` function takes the following parameters, from left to right
  - Artist dictionary
  - Current list of artists
  - Activity to filter on
  - Boolean `include` indicating whether artists associated with `activity` should be included or excluded
- It should return a **new** list that results from filtering. Let's write it!