Introduction to:
Computers & Programming:
Dictionaries and Sets

Adam Meyers
New York University
Outline

• Unordered Collections of Objects
  – Sets (Brief discussion)
  – Dictionaries (Main focus)
• Details
  – Definitions
  – Syntax
  – Methods and Functions
  – Usage
• Problems using dictionaries
What is a Set?

• A set is an unordered collection of objects
  – Long history in math with definitions for various operations and relations: subset, superset, intersection, union, etc.

• Python's implementation of sets
  – No indices (sets are unordered)
  – No duplicate objects (there is no way to distinguish a 2\textsuperscript{nd} instance of the same item)
  – mutable (like lists)
    • Immutable version of sets: \texttt{frozenset}

• Sets vs Lists
  – Sets are better models of some types of problems than lists
  – Lists are more versatile and can usually handle set-oriented problems, with a few modifications, e.g., if duplicate elements are eliminated.

• I will not require sets in the homework or in tests (though if you choose to use them to solve a problem, that is OK)
Set Syntax, Methods, Functions

• Curly brackets around elements with commas in between
  – my_set =\{1,2,3\}

• Adding elements (.add and .update)
  – my_set.add(4) ## duplicate elements do not get added

• Removing an element
  – my_set.discard(9) ## no error if element is not there
  – my_set.remove(9) ## error if element is not there

• Set operations from math
  – set1.union(set2)
  – set1.intersection(set2)
  – set1.difference(set2)
More Set Syntax, Methods, Functions

- Boolean tests for membership, subset, superset
  - 1 in set1 ## membership
  - set1.issubset(set2)
  - set1.issuperset(set2)
- len(set1) → number of elements
- loop for item in set: ## similar to list, order is not important
  print(item)
- set(sequence) ## coerces sequences into a set
Dictionaries

• Syntax – instances of key + : + value, separated by commas
  – Keys must be immutable: string, number, tuple, etc.

• A dictionary can be initialized as follows:
  my_dictionary = {'eat': 'verb', 'book': 'noun', 'orange': 'adjective'} # with values
  my_dictionary2 = {} # empty

• Items can be looked up in a dictionary, as follows:
  my_dictionary['eat']
  my_dictionary.get('eat')
  – If key is not in dictionary, these give error
  – Keys in dictionaries function similarly to the way indices function in lists (lookup, setting values)

• To check if item is in dictionary:
  'eat' in my_dictionary

• Items can be added or changes as follows:
  my_dictionary2['book'] = 'noun'

• Dictionaries are mutable (like lists)
  – Changing a variable pointing to a dictionary changes the dictionary
Dictionaries 2

• What is a dictionary in Python?
  – Dictionaries consists of sets of key/value pairs
    • For example, the key 'book' currently has the value 'noun'
  – Not ordered. Look up by keyword, not by index as with sequences.

• Why have a dictionary?
  – To store fairly large amounts of information that has no natural order.

• Examples
  – Words: part of speech (noun, verb, etc.), definitions, pronunciation, etc. (Lexicons)
  – Sports player: statistics
  – Food: ingredients
  – Medicine: side effects
  – Telephone books (or reverse telephone books)
Dictionary Methods and Functions

- Methods for displaying contents of dictionary:
  - `dictionary.items()`, `dictionary.keys()`, `dictionary.values()`

- Functions for changing dictionary:
  - `dictionary.clear()` – erases contents
  - `my_dictionary.update([[\'bear\',\'common_noun\'],[\'friendly\',\'adjective\']])` – argument can be sequence or dictionary – all pairs are added
  - `my_dictionary.pop(\'book\')` removes item/returns value
  - `my_dictionary.popitem()` removes/returns arbitrary key/value pair

- Copying: `my_dictionary.copy()`

- `len(dictionary)` → number of elements in a dictionary

- `for item in dictionary:`  ## for loop treats dictionary as list of keys
  - `print(item, dictionary[item])`

- `'book' in dictionary`  ## True if 'book' is a key in dictionary; False otherwise
Dictionaries are Hash Tables

- The dictionary data structure is Python's implementation of a hash table, a popular data structure for mapping keys to values.
- There is a special “hash” function which maps keys to (ideally) unique places in memory. This makes it possible to “look up” the values efficiently, even though the items are not stored in sequential order.
  - Example (pretending that only sorted strings are considered and ignoring efficiency)
    - assign each character a prime number
    - assign each character length a prime number
    - for strings of multiple characters, multiply these prime numbers together
- Due to efficiency concerns, most hash functions do not map keys uniquely, but use heuristics (compromises) to get around this problem.
- Hash function are derived from keys – this can only work if the key doesn't change (different keys have different hash values).
  - Immutable objects can be hash keys (strings, numbers, tuples, etc.)
  - Mutable objects (lists, dictionaries, etc.) cannot be hash keys
  - Imagine trying to looking up a word in a physical (book) dictionary while somebody added and subtracted letters from the beginning and end of the word. Since the dictionary is in alphabetical order, its position in the dictionary would keep changing and you would have trouble finding it.
- For details see (for example) Wikipedia's entries for hash table and hash function.

Intro to: Computers & Programming: Dictionaries
CSCI-UA.0002
Class Exercise

1. Set a global variable called telephone_book to an empty dictionary
   
   telephone_book = {}

2. Write a simple function for users to add phone numbers into telephone_book. Remember to add key+value to dictionary, the command is:
   
   dictionary[key] = value

   The keys should be names (strings) and the values should be strings consisting of numbers and hyphens, e.g., '212-123-4567'

3. Add several names (at least 5) and phone numbers to your dictionary using the function you wrote.

4. Write a function to save the phonebook in a text file

5. Revise the program so it can handle cases where one name is associated with multiple phone numbers
Random Sentence Generation Program

• Demonstration of the program
• Where the words came from
  – 1000 most common words according to: [http://www.bckelk.ukfsn.org/words/uk1000.html](http://www.bckelk.ukfsn.org/words/uk1000.html)
    • Sorted them automatically into: nouns, plural_nouns, adjectives, verbs, adverbs
    • according to some dictionaries
    • also edited to work OK for simple sentences
  – Added some proper nouns also (person names, organization names, etc.)
What the Sentence Generator Does

- Generates sentences according to a set of rules (aka, a grammar).
- These rules are called phrase structure rules, commonly used in both linguistics and computer science.
- The rules consist of two types of symbols: Terminals & Nonterminals.
- Each rule consists of a symbol (a Nonterminal) followed by an arrow and a list of symbols (Terminal or Nonterminal).
- The interpretation is that the symbol before the arrow can be replaced by the symbols following the arrow.
- Terminal symbols can be replaced with random words of that type.
- The system automatically does this replacement until a list of words results.
Rules Used by Sentence Generator

- $S \rightarrow NP \ VP$
- $NP \rightarrow \text{DetP ADJ\_SEQ NOUN\_SEQ}$
- $NP \rightarrow \text{DetP NOUN\_SEQ}$
- $NP \rightarrow \text{ADJ\_SEQ plural\_noun}$
- $NP \rightarrow \text{plural\_noun}$
- $NP \rightarrow \text{Proper\_noun}$
- $\text{DetP} \rightarrow \text{NP pos}$
- $\text{DetP} \rightarrow \text{determiner}$
- $\text{pos} \rightarrow 's$
- $\text{PP} \rightarrow \text{preposition NP}$
More Rules Used by Sentence Generator

- $\text{ADJ\_SEQ} \rightarrow \text{adjective}$
- $\text{ADJ\_SEQ} \rightarrow \text{adjective} \ \text{ADJ\_SEQ}$
- $\text{NOUN\_SEQ} \rightarrow \text{plural\_noun}$
- $\text{NOUN\_SEQ} \rightarrow \text{singular\_noun}$
- $\text{VP} \rightarrow \text{verb}$
- $\text{VP} \rightarrow \text{verb} \ \text{NP}$
- $\text{VP} \rightarrow \text{verb} \ \text{NP} \ \text{PP}$
- $\text{VP} \rightarrow \text{adverb} \ \text{verb}$
- $\text{VP} \rightarrow \text{adverb} \ \text{verb} \ \text{NP}$
- $\text{VP} \rightarrow \text{adverb} \ \text{verb} \ \text{NP} \ \text{PP}$
Some Linguistic Background

• A simple sentence is:
  – A Noun Phrase (the subject) plus a Verb Phrase
• A Verb Phrase is a verb plus a bunch of other phrases, mostly following it.
• A Noun Phrase is:
  – a determiner (optional for plurals) plus some adjectives, plus a noun (and possibly other stuff)
  – or a name (which can consist of multiple words)
• This is a simplified view of things, but sufficient for understanding this program.
• The program will produce sentences that are nearly grammatically well-formed sentences (some will be OK).
• It also produce mostly semantically strange sentences.
Dictionary Lookup for Rules

- The dictionary maps each symbol before the arrow in the rules to each of the list of the possible sequences following the arrows.

- For example, the entry for 'NP' is the following list:
  - [['DetP','ADJ_SEQ','NOUN_SEQ'],['DetP','NOUN_SEQ'],['NP','PP'],['ADJ_SEQ','plural_noun'],['proper_noun'],['plural_noun']]  

- Each terminal symbol (\textit{singular\_noun, plural\_noun}, etc.) is linked in the dictionary to a list of all the words of that class.
Generating a Random Sentence

• To generate a random sentence, keep replacing non_terminal symbols with terminal symbols until the whole phrase consists of words. Then print out those words.

• Start by looking up S (sentence) in the dictionary
  – There is only one expansion: $S \rightarrow NP \, VP$
  – The system starts with the “stack” set to ['VP', 'NP']
  – Right most item (top of the stack) is popped and looked up in the dictionary: 1 of the 4 possible expansions is chosen randomly and added back in the stack, e.g., the stack may now be: ['VP', 'NOUN_SEQ', 'ADJ_SEQ', 'DetP']
  – The loop repeats, this time the next item 'DetP' is looked up.
  – Any time a word is output, it has no dictionary entry and it gets added to a new list called result. In the end we have a list of words, which is essentially the output.

• Demo with trace
Sample Program: Movie Rating System

• Problem: The current MPAA movie rating system is broken (see, for example, the documentary *This Film is Not Yet Rated*)
  – Movies are, in effect, censorted by a bunch of extremists who do not really represent the population of the U.S.

• The system is an attempt at an alternative rating system designed to help people screen movies by what they actually find offensive. As different people find different things offensive, the system is complex and does not translate into a small number of classes. Furthermore, you are crowd-sourcing movie ratings to the actual population, not a few anonymous raters who are not accountable to the public.
Movie Rating System 2

• The Pilot Program uses an initial list of 34 traits of things that might offend somebody. Users can also add offensive traits of their own.

• Users have the option of choosing any number of trait and rating a movie from 1 to 10 with respect to its offensiveness.

• The program compiles this information in an organized way. Later versions of the program may use a standard set of traits.
Representation: a dictionary with lists of dictionaries as values

• **review_dictionary** is a dictionary of all ratings
  – Indices are names of movies
  – Values are lists of reviews for each movie
  – It can be initialized by the reviews in a file
  – Users can add reviews
  – The information can be saved in a file
  – **review_dictionary** should be a dictionary rather than a list because we cannot know which movies will be reviewed ahead of time
  – The value could be a dictionary, rather than a list if each user was associated with a key (an id number) and we wanted to prevent a user from reviewing the same movie more than once.

• Each review in the list of reviews is a dictionary
  – The keys are offensive traits and the values are ratings from 1 to 10 indicating how offensive
  – A review should only contain one rating for each trait
  – A review only needs to use a subset of the traits
  – The number of possible traits may change, at least during the experimental stage
File I/O

- File Format
  - Each line is of the form:
    - Movie_Name + tab + feature:value + tab + feature:value...
    - Jurassic Park  'Physical Appearance Stereotype':8  'Inaccurate Science':9
  - Each line represents one review

- Writing to a file
  - Loop through the dictionary
    - For each key in the dictionary
      - For each item in the value of that key
        » Write out the key followed by feature value pairs
        » Feature value pairs are separated by a tab
        » Features are separated from values by ': '

- Reading from a file
  - For each line, update the dictionary:
    - Split the line at tabs, the first item is the key and the rest is the entry
    - The review is a dictionary based on the rest of the entry
      - Split each item in the entry at “: “ to form keys and values
      - For each key, value, do entry[key] = value
    - If the key is already in the dictionary, append this review to the value
    - Otherwise add a new entry to the dictionary for this key, consisting of a list containing this review
Summary

• Sets and Dictionaries are both unordered collections of items
  – They are mutable
  – Items in sets and dictionaries are unique (unlike lists)
• Sets model the math object – not a main focus of this class
• Dictionaries provide a way of looking up items without reference to sequences
  – Even though some dictionaries could be ordered (words in alphabetic order), this would be inconvenient/inefficient for very large sequences
  – Other dictionary keys have no natural order
  – Useful for modeling systems requiring arbitrary lookup
• Modeling actual problems may require data structures that combine the various data types that we have covered in this class
Homework Reading

- Read Chapter 8 in
  - *Common Lisp: A Gentle Introduction to Symbolic Computation* by David S. Touretzky
  - The pdf can be found at:
  - Read the stories about the dragon, but instead of reading about the lisp code (sections 8.3, 8.5 and 8.7) look at the corresponding python code (see comments) at:
    - http://cs.nyu.edu/courses/fall15/CSCI-UA.0002-007/dragon_recursion.py
- Read Chapter 12 in Gaddis Book
Homework Question 1: Make a Backwards Phonebook

• Write a function `make_backwards_phonebook` that:
  – Reads in a phonebook file in the format that we created in class (a sample file on the website)
  – Stores it in a dictionary
  – Creates a new dictionary using the phone numbers as keys and the names as values
  – Writes new dictionary in a file (.csv, .tsv, or record structure you define)

• The function should take an input filename and an output filename as parameters
  – `make_backwards_phonebook(phbook_file, back_phbook_file)`

• Write a function `user_input_make_bw_phonebook` that:
  – Uses `input` statements to get input and output filenames
  – Checks if the input file exists and handles the exception elegantly if it does not
  – Calls `make_backwards_phonebook` using the user input as parameters
Homework Question 2: Create program that replaces holidays with dates

1. Create a dictionary of 8 or more holidays for 2015 listing dates as strings of the form '12/31/2015' (e.g., New Years Eve)

2. Write a function that uses the dictionary to replace holidays in a string with the corresponding dates. The function should work as follows:
   
   (a) It should split the string into words (split at spaces)

   (b) It should check if each word is in the dictionary and replace it in the list with its value, e.g., the list ['I','wish','you','a','merry','Christmas'] should become: ['I','wish','you','a','merry','12/25/2011']

   (c) Convert this list into a string using a for loop. Start with the empty string and add the items in the list, separated by spaces. Return the resulting string.

   (d) Your function in (b) should deal with punctuation and capitalization in some way. For capitalization, you could simply convert to lower case and only store lowercase items in your dictionary. For punctuation, you could strip off characters that are periods or commas. Smoother ways of handling these issues are worth more points.
Homework Question 3: Word Freq Dictionary

• Write a program that reads in a text file and counts instances of each word, recording those instances in a dictionary. Then it should print out that information in a new file.

• The program should:
  – Read in each line from the input file
  – Split each line into words by spaces
  – Remove periods, question marks, commas, and periods from the end of each word using the string.trim method
  – Remove quotation marks ("",""') from beginning and end of words
  – Count the occurrences of the words using a dictionary
  – Write out the dictionary information into a file in a consistent format (e.g., word: number, .tsv, .csv, etc.)
Extra Credit: Day in the Future Problem from Earlier in the Semester

• Look at the lecture from the 3rd class of the semester.

• Given
  – Today's date:
    • Month, day, year, and day of the week
  – Any number of days in the future: 1, 1500, 50,000, …

• Identify the new day:
  – Month, day, year and day of the week

• This should be possible using lists and dictionaries.

• You should also be able to account for leap years
  – Either assume every 4th year is a leap year or explain what the more complex assumption is in a comment.
Complex Record Structures

• Data structures containing Data Structures
  – Lists inside of Lists
  – Dictionaries with List values
  – Lists of Dictionaries
  – Dictionaries with Dictionary Values
  – Dictionary with Lists of Dictionaries as values
  – Etc.

• Design data structure based on the problem
  – Lists are sequences of items
  – Dictionaries are unstructured items with quick lookup by key (e.g., a word or phrase)