Nicholas Bowman, Sonja Johnson-Yu, Kylie Jue **Section #4**
CS106AP **July 17, 2019**

Section Handout #4: Parsing, Dictionaries, and Nested Data Structures

1. Introduction to String Parsing

Write code and a couple of appropriate doctests for each of the following functions. The Section 4 starter code and problem description contains some suggestions for tests but does not provide an exhaustive list. All of these functions are solvable using the string fundamentals we've seen in lecture, including literals, built-in functions (like len() and str.find()), indexing and slicing, and string concatenation.

a. **`remove_front(s)`**: Given a string **`s`**, if the string is length 2 or more, return it without its first 2 characters. Otherwise, return the string unchanged.
   - Expected behavior
     - **`remove_front('a') → 'a'`**
     - **`remove_front('MrKrabs') → 'Krabs'`**

b. `x_to_end(s)`: Given a string **`s`**, if the string contains one or more occurrences of the letter 'x,' return the substring starting from the first occurence of the letter 'x' to the end. If the string doesn't contain the letter 'x,' return the empty string.
   - Expected behavior
     - **`x_to_end('These are not the letters you are looking for.') → ''`**
     - **`x_to_end('On the map, x marks the spot') → 'x marks the spot'`**
   - Bonus: How could you modify this function so that it could look for any letter, not just 'x'? (This is called "generalizing" the function!)

c. `simile_builder(s)`: Given a string **`s`**, which may or may not contain a colon, build a simile from that string. If the string does not contain a colon, return a sad message of your choosing. If the string contains a colon, extract two substrings. The first should be the substring starting from the beginning of the string and going up to but not including the colon. The second should be the substring starting after the colon and going to the end of the string. You should return a string that is in the form **`'a x is like a y'`**, where **`x`** and **`y`** are the first and second extracted substrings.
   - Expected behavior:
     - **`simile_builder('Bah Humbug') → 'Couldn't make a simile, sad times.'`**
     - **`simile_builder('dog:best friend') → 'a dog is like a best friend'`**
   - Bonus: How can you modify this code to follow correct grammar rules? That is, if the substring **`x`** begins with a vowel, then it should be preceded by 'an' instead of 'a.'

d. **`exclaim(s)`**: Given a string **`s`**, look for the first exclamation mark. If there is a substring of 1 or more alphabetic characters immediately to the left of the exclamation mark, return this substring including the exclamation mark. Otherwise, return the empty string.
   - Expected behavior:
     - **`exclaim('xx Hello! yy') → 'Hello!'`**

        - **`exclaim('Nothing ! to exclaim')`** → **`''`**
5. **`vowels(s)`:** Given a string **`s`**, look for the first colon. If there is a substring of 1 or more vowels immediately to the right of the colon, return this substring without the colon. Otherwise, return the empty string.
    - Expected behavior
        - **`vowels('victory screech:aieee!')`** → **`'aieee'`**
        - **`vowels('question:why?')`** → **`''`**

2. Extracting Email Hostnames

Now, we're going to turn our attention to a parsing task we'd be more likely to see in the real world: parsing email addresses. For the purposes of this problem, we'll be using a simplified format of an email address as follows:

`username@hostname`

where **`hostname`** is a string with at least 4 characters. It consists of at least one period and only alphabetic characters. In addition, the username can be any length, including 0 characters. Some examples are:

```
kyliej@stanford.edu # valid email address
nbowman@cs.stanford.edu # valid email address
jillian@website     # invalid email address, needs at least one period
sheridan@email1.com # invalid, since 1 isn't a letter or period
@gmail.com          # valid email address
sam@a.b             # invalid, less than 4 characters long.
```

Suppose you have a file called **`emails.txt`** that looks like this:

```
Please forward this email to ingrid@stanford.edu for me. Thanks!
Can someone tell me who owns the parth@yahoo.com email address?
The email brahm@gmail.com keeps sending me spam mail.
Please forward this email to justin@stanford.edu for me. Thanks!
Omg @ye is my favorite!
Do you think a@b.c is spam?
This one isn't spam: a@d.tv
Hello, world!
Why am I getting emails from trey@spam.com?
```

which has at most one email address per line of the file. Your job is to write the following function:

`def extract_all_hostnames(filename)`

which takes in a string representing a file's name and returns an **alphabetically sorted list** of all the **unique hostnames** in the file. For example, calling the function with the parameter **`emails.txt`** would have the following result:

```
>>> extract_all_hostnames('emails.txt')
['d.tv', 'gmail.com', 'spam.com', 'stanford.edu', 'yahoo.com']
```

In writing this function, think about how best to decompose it into functions that are responsible for subparts of the problem. For example, consider implementing a function which extracts a hostname from a single line and think about how you might use it.

3. Practice with Dictionaries

- **`def int_counts(ints)`**: Given a list of integers, create and return a dict that stores the counts of the elements in the list: that is, each unique integer in **`ints`** is a key in the dictionary and the corresponding value is the number of times that integer appeared in the list.
    - Expected behavior:
        - **`int_counts([1, 2, 1, 4])`** → **`{1: 2, 2: 1, 4: 1}`**
        - **`int_counts([])`** → **`{}`**
- **`def mutual_friends(phonebook_one, phonebook_two)`**: In the days before Facebook, one of the easier ways to find mutual friends was to compare address books and find common entries. Write a function named **`mutual_friends()`** that accepts as parameters two dicts from strings (keys) to integers (values) representing two phonebooks and returns a new dict containing only the key/value pairs that exist in both of the phonebooks. Remember that for an entry to be included in your result dict, not only do both dicts need to contain that key, but they must also both map that key to the same value (phone number). For example, consider the following two dicts:

  ```
  {'Jenny':8675309,        'Julia':2124320,        'Kylie':4602121,
  'Sonja':4444444, 'Nick':8080543}

  {'Logan':6202121,         'Jeff':8888888,         'Nick':8080543,
  'Kylie':4602121, 'Sonja':4444444, 'Jenny':2128765}
  ```

  Calling your function on the preceding dicts would return the following new dict (the order of the key/value pairs does not matter):

  ```
  {'Kylie':4602121, 'Nick':8080543, 'Sonja':4444444}
  ```

  Notice how even though the key **`Jenny`** is present in both dicts, the contact is not included in our final result dict because it maps to a different phone number in each phonebook.

4 . Tracing Your Way Through The Office

This problem is great practice for the midterm.

Consider the following program:

```
def dundees(pam, michael, dwight):
    jim = 5
    for i in range(1, pam, 2):
        jim += i
        if michael:
            battlestar_galactica(pam, jim)
            jim = scotts_tots(jim, pam, dwight)
        else:
            if not dwight:
                jim = scotts_tots(pam, jim, not michael) #  'not'
just flips a boolean from True to False or vice-versa
                battlestar_galactica(jim, 42)
    return jim // pam

def battlestar_galactica(toby, kevin):
    creed = toby
    toby = kevin
    kevin = creed

def scotts_tots(oscar, angela, jim):
    kelly = 2 * oscar
    if jim:
        kelly -= angela
    return kelly

def main():
    args = sys.argv[1:]
    michael = False
    holly = False
    if '-scott' in args:
        michael = True
    if '-flax' in args:
        holly = True
    mifflin = int(args[0])
    print(dundees(mifflin, holly, michael))

if __name__ == '__main__':
    main()
```

For each of the following calls, trace through the program's execution to determine what the program prints:

```
$ python3 the-office.py 2 -flax

$ python3 the-office.py 5 -scott

$ python3 the-office.py 4
```

Note: The material covered in problems 5-7 will not be tested on the midterm.

5. Nested List Functions

a. **`threes(n)`**: Given a non-negative integer **`n`**, create and return a list of **`n`** lists, each of which has length 3. Each inner list should contain 3 consecutive integers. The first inner list should start with a 1, and every subsequent inner list's first element should increase by 1.
   - Expected behavior:
     i. **`threes(4)`** → **`[[1,2,3], [2,3,4], [3,4,5], [4,5,6]]`**
     ii. **`threes(1)`** → **`[[1, 2, 3]]`**

b. **`countdown(n)`**: Given a non-negative integer **`n`** that is less than or equal to 10, create and return a list of **`n`** lists, where each inner list counts down from 10 to successively smaller numbers.
   - Expected behavior:
     i. **`countdown(3)`** → **`[[10], [10, 9], [10, 9, 8]]`**
     ii. **`countdown(5)`** → **`[[10], [10, 9], [10, 9, 8], [10, 9, 8, 7], [10, 9, 8, 7, 6]]`**
     iii. **`countdown(0)`** → **`[]`**

6. Nested Dictionary Functions

a. **`def first_list(strs):`** Given a list of strings **`strs`**, create and return a dictionary whose keys are the unique first characters of the strings in **`strs`**. The value for each key should be a list of the strings from **`strs`** that begin with that character, in the same order that they appear in **`strs`**.
   - Expected behavior:
     i. **`first_list(['aaa', 'abb', 'ccc'])`** → **`{'a': ['aaa', 'abb'], 'c': ['ccc']}`**
     ii. **`first_list(['aaa', 'abb', 'xcc', 'x', 'acc'])`** → **`{'a': ['aaa', 'abb', 'acc'], 'x': ['xcc', 'x']}`**
     iii. **`first_list([])`** → **`{}`**

b. **`def suffix_list(strs)`**: Given a list of strings **`strs`**, create and return a dictionary whose keys are the unique two-letter suffixes of the strings in **`strs`**. The value for each key should be a list of the strings that end with that suffix, in the same order that they

appear in **strs**. A suffix is defined as the last 2 characters of a string, and a string that is less than 2 characters long has no suffix.

- Expected behavior:
  i. **suffix_list(['hello', 'mellon', 'mello', 'felon', 'x', 'xyz']) →{'lo': ['hello', 'mello'], 'on': ['mellon', 'felon'], 'yz': ['xyz']}**
  ii. **suffix_list([]) → {}**
  iii. **suffix_list(['zab', 'x', 'y', '']) → {'ab': ['zab']}**

c. def reverse_dict(animal_dict): Suppose we have a **dict** called **animal_dict** that associates people with their favorite animals. For example, consider the following example dictionary:

```
{'Kylie': 'Mantis Shrimp', 'Nick': 'Mantis Shrimp', 'Elena': 'Dog', 'Laikh': 'Parrot', 'Laura': 'Cat'}
```

This dictionary is useful for telling us which animal a particular person likes. For example, **animal_dict['Nick']** would evaluate to the string **'Mantis Shrimp.'** However, it is less useful for telling us which people like a particular animal. In order to do that, we need some way of "reversing" the dictionary, so that we can look up **'Parrot,'** for example, and find out that **'Laikh'** likes parrots. Your job is to write the following function:

```
def reverse_dict(animal_dict)
```

which takes in a dict such as the one described above and prints which people like which animal. For example, for the dictionary above our output might look like this:

```
Cat: Laura
Dog: Elena
Mantis Shrimp: Kylie, Nick
Parrot: Laikh
```

Note that multiple people can like the same animal, and thus your main challenge is to figure out a way to represent multiple people associated with the same animal. Matching the exact format of the program above isn't particularly important: the point of this problem is to figure out how you can leverage the data structures we've been discussing in class recently to solve a problem like this!

7. Big Tweet Data

In this program, you'll write a program that reads through a large collection of tweets and store the data to keep track of how hashtags occur in tweets. This is a great example of how Python can be used in data analysis tasks.

Our Dataset

For the purposes of this problem, each tweet is represented as a single line of text in a file. Each line consists of the poster's username (prefixed by a '@' symbol), followed by a colon and then the text of the tweet. Each character in this file can be a character from any language, or an emoji, although you don't need to do anything special to deal with these characters since they're also represented as unicode! One such file in the PyCharm project we provide is **`small-tweets.txt`**, which is reproduced here:

```
@BarackObama: Missed President Obama's final #SOTU last night? Check
out his full remarks. https://t.co/7KHp3EHK8D
@BarackObama: Fired up from the #SOTU? RSVP to hear @VP talk about
the work ahead with @OFA supporters:
https://t.co/EIe2g6hT0I https://t.co/jIGBqLTDHB
@BarackObama: RT @WhiteHouse: The 3rd annual #BigBlockOfCheeseDay is
today! Here's how you can participate:
https://t.co/DXxU8c7zOe https://t.co/diT4MJWQ…
@BarackObama: Fired up and ready to go? Join the movement:
https://t.co/stTSEUMkxN #SOTU
@kanyewest: Childish Gambino - This is America
https://t.co/sknjKSgj8c
@kanyewest:              https://t.co/KmvxIwKkU6
@GonzalezSarahA: RT @JacobSmithVT: Just spent ten minutes clicking
around this cool map #education #vt #realestate
https://t.co/iqxXtruqrt
```

We provide 3 such files for you in the PyCharm Project: **`small-tweets.txt`**, **`big-tweets.txt`** and **`huge-tweets.txt`**.

Building a `user_tags` Dictionary

Central to this program is a **`user_tags`** dictionary, in which each key is a Twitter user's name like **`'@BarackObama.'`** The value for each key in this dictionary is a second, nested dictionary which counts how frequently that particular user has used particular hashtags. For example, a very simple **`user_tags`** dictionary might be:

`{'@BarackObama': {'#SCOTUS': 4, '#Obamacare': 3}}`

We'll explore this dictionary in some more detail as we go through this problem, but as a matter of nomenclature, we'll call the inner dictionary the '**hashtag_counts**' dictionary (since it keeps track of the hashtag counts for each user). Our high-level strategy is to change the '**hashtag_counts**' dict for each tweet we read, so it accumulates all the counts as we go through the tweets.

% c gy t |w x|kz{pvuz!

Given the dictionary above, what updates we would make to it in each of the following cases?

- We encounter a new tweet that reads '**@BarackObama: #Obamacare signups now!**'
- We encounter a new tweet that reads '**@kanyewest: https://t.co/KmvxIwKkU6**'

& Ut wsk t ku{pun **add_tweet()**

The **add_tweet()** function is the core of this whole program and is responsible for performing the update to a **user_tags** dictionary described above. The tests shown below represent a sequence of calls to the **add_tweet()** function that might be used to build up the **user_tags** dictionary over time, expressed as a series of doctests. For each doctest, you can see the dictionary that is passed in to the function call, and the dictionary that the function call returns is on the next line. The first test passes in an empty dictionary (**{}**) and the string '**@alice: #apple #banana**', and the function returns a dictionary with 1 user and 2 tags in the user's '**hashtag_counts**' dict. The second test then takes that returned dictionary as its input, and so on. Each function call adds more data to the **user_tags** dictionary.

You're all string parsing experts by now, so we won't make you do that work anymore. We've provided you with two functions: **parse_tags(tweet)** and **parse_user(tweet)**. **parse_tags()** takes in a tweet and returns a list of tags in the tweet. **parse_user()** takes in a tweet and returns the username of the user who posted the tweet.

Your job is to implement the **add_tweet()** function using the provided **parse_tags(tweet)** and **parse_user(tweet)** functions:

```
def add_tweet(user_tags, tweet):
    """
    Given a user_tags dict and a tweet, parse out the user and tags,
    and add those counts to the user_tags dict. Return the updated user_tags
    dict.
    If no user exists in the tweet, return the user_tags dict unchanged.
    Note: Call the parse_tags(tweet) and parse_user(tweet) functions to pull
    the parts out of the tweet.
    >>> add_tweet({}, '@alice: #apple #banana')
    {'@alice': {'#apple': 1, '#banana': 1}}
    >>> add_tweet({'@alice': {'#apple': 1, '#banana': 1}}, '@alice: #banana')
```

```
    {'@alice': {'#apple': 1, '#banana': 2}}
    >>> add_tweet({'@alice': {'#apple': 1, '#banana': 2}}, '@bob: #apple')
    {'@alice': {'#apple': 1, '#banana': 2}, '@bob': {'#apple': 1}}
    """
```

'‿ U t wsk t ku{pun **`parse_tweets()`**

Use **`add_tweet()`** in a loop to build up and return a **`user_tags`** dict. This should look mostly like other file-reading functions you've written, and your job is to make sure you understand how to follow the pattern of creating and updating a dictionary with the help of the **`add_tweet()`** function. Restated, the responsibility of **`add_tweet()`** is to update a dictionary, and **`parse_tweets()`** must create and maintain that dictionary as it is updated.

Running your program

We provide a **`main()`** function that calls the **`parse_tweets()`** function you implemented in a variety of ways. To use it, run the program from the terminal. Run with just 1 argument (a data file name), it reads in all the data from that file and prints out a summary of each user and all their tweets and counts:

```
$ python3 tweets.py small-tweets.txt
@BarackObama
  #BigBlockOfCheeseDay -> 1
  #SOTU -> 3
@GonzalezSarahA
  #education -> 1
  #vt -> 1
  #realestate -> 1
```

When run with the **`'-users'`** argument, **`main`** prints out all the usernames:

```
$ python3 tweets.py -users small-tweets.txt
users
@BarackObama
@kanyewest
@dog_rates
@GonzalezSarahA
```

When run with the **`'-user'`** argument followed by a username, the program prints out the data for just that user.

```
$ python3 tweets.py -user @BarackObama small-tweets.txt
user: @BarackObama
 #BigBlockOfCheeseDay -> 1
 #SOTU -> 3
```