CIS 41B - Lab 4: web api, multithreading, multiprocessing, os, review of json, gui

Since summer vacation is right around the corner (if not already here) for many students, write an application that lets the user look up information on US national parks so that they can plan their summer vacation trips.

The lab consists of 2 files: lab4thread.py and lab4process.py. Both files run independently of each other and both files provide similar results.  
Follow the steps below in sequence to build up your lab4thread.py file first. Then convert the multithreading part to multiprocessing to produce lab4process.py.

Part 1: web access

A. The data for all US national parks is from the National Park Service (NPS), and the NPS has provided an API for developers to access data at: <https://www.nps.gov/subjects/developer/index.htm>

To use the API, you need to register for an API key. At the page above, click on the "Get Started" link to register and receive a key (a long sequence of alphanumeric characters) through email.

B. To access data, use requests as with lab 3 but the request string is the API URL.

As you can see from the documentation at the developer page, there are many ways to access data from the API. For this lab we want to let the user look up parks within a state, so the request string is:

"https://developer.nps.gov/api/v1/parks?stateCode=<state code>&api\_key=<your key>"

where <state code> and <your key> are fields that you need to fill in.

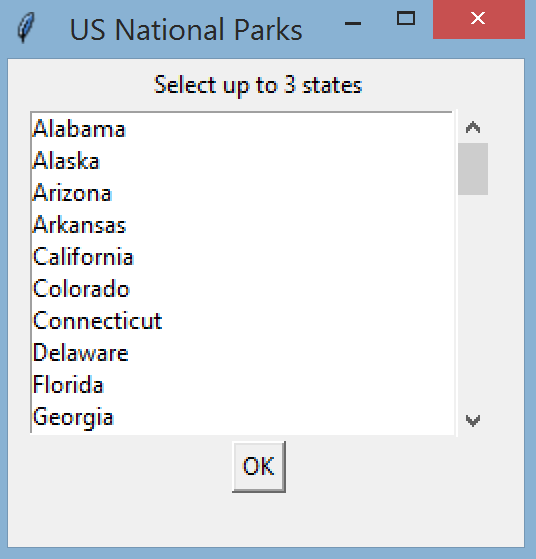
C. To look up the state code, see the file states\_hash.json, which has a dictionary of state codes as keys, and state names as values. [Source: <https://gist.github.com/mshafrir/2646763>]

Part 2: lab4thead.py

The lab4thread.py has 2 GUI classes: a main window class and a display window class.

The **main window** class does the following tasks:

1. Display a window with 5 widgets: a prompt label, a listbox with a vertical scrollbar, an OK button, and a blank label

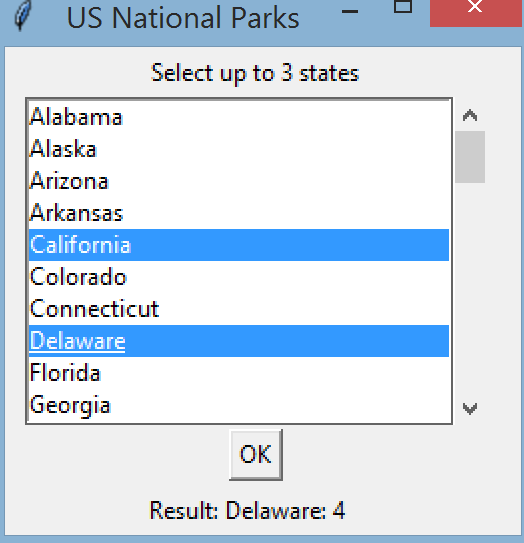
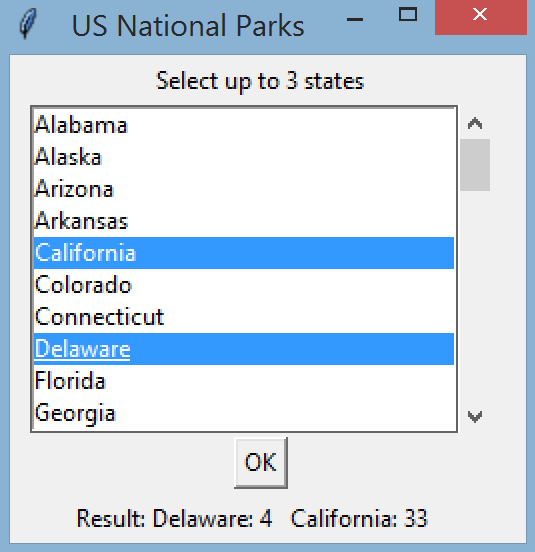


* + The prompt label asks the user to select up to 3 states from the listbox
  + The listbox can display 10 states, and it has a scrollbar so the user can scroll through all 50 states. The state names come from the states\_hash.json file.
  + The user clicks the OK button to make their choice(s) of states
  + The blank label is for status messages later

When this window appears, the user cannot selec the main window.

1. If the user clicks OK, check that there are 1 - 3 choices, inclusive, from the listbox.  
   If there are no choices or more than 3 choices, pop up an error window to display the error message. Then let the user choose from the states again.
2. If the user makes 1-3 choices, for each choice:
   * Create a thread to fetch data for the state that the user chooses.  
     [Hint: the user chooses a state name, and the states\_hash.json file has the equivalent state code]
   * As each thread finished fetching the data, use the blank label at the bottom of the window to display the state name and the number of parks there are in the state.  
     Note that the status message for each state should show up *as each thread is finished*, don't print all the status messages at the very end when all threads are done.

In the example below, the user selects California and Delaware:

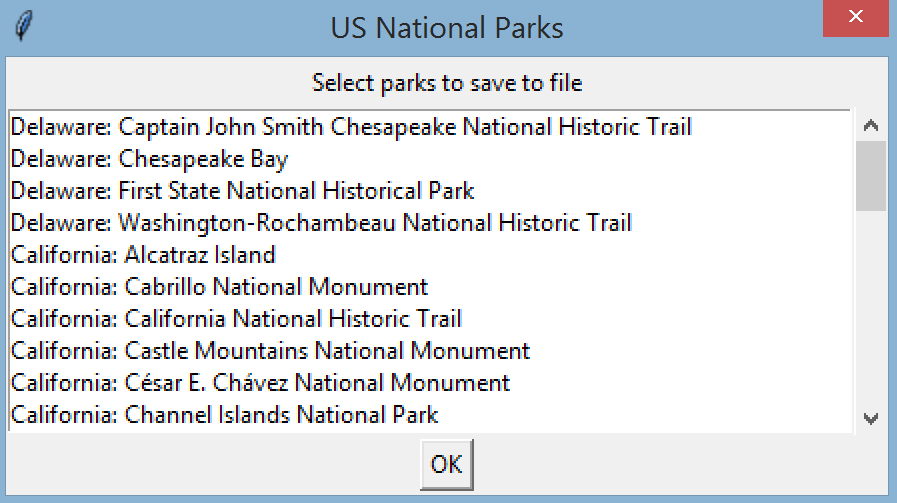
Delaware results came back first California results came back after

and is displayed first and is displayed second

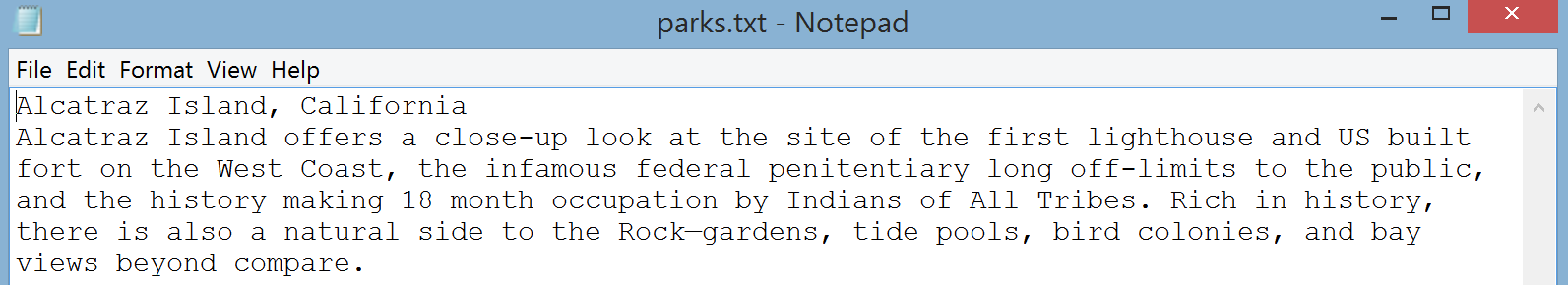
1. When the results from all the selected states have come back, the main window calls a display window to display all the results.

The **display window** class does the following tasks:

1. Display all the results in a listbox with the format: state name : park full name  
   and have a label to prompt the user to select the park(s)



1. If the user clicks OK, check that there is at least 1 choice from the listbox.  
   If there are no choices, pop up an error window to display an error message. Then let the user choose from the parks again.
2. If the user makes at least 1 choice, pop up a file dialog window to let the user choose a directory to save the output file. The default directory of the file dialog window is the current directory.
3. If the user chooses a directory, check to see if a file named "parks.txt" already exists in the user chosen directory.  
    If it does, pop up a warning window to let the user know that the file will be overwritten.  
   Then write to the "parks.txt" file the "fullName" field, the state name, and the "description" field of each of the selected park's data. The fullName and state name is on one line, and the description should be written on a second line.  
   Here's an example for a national park in California when the output file is opened:



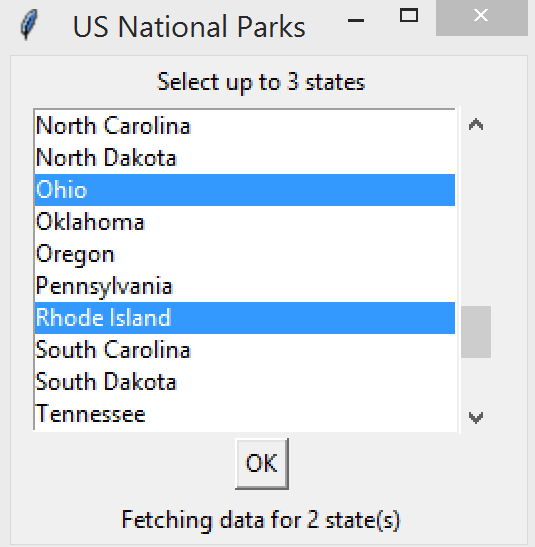
1. When the file has been written, the display window closes and the user is back at the main window.
2. If the user chooses to cancel from the file dialog window (instead of choosing a directory), then the user is back at the display window again and can select parks from the listbox.
3. If the user choose to close the display window by clicking 'X', then the user is back at the main window.
4. When the user is back at the main window (from clicking 'X' or from saving data to file), the status label at the bottom of the main window clears out and becomes blank again.

Part 3: lab4process.py

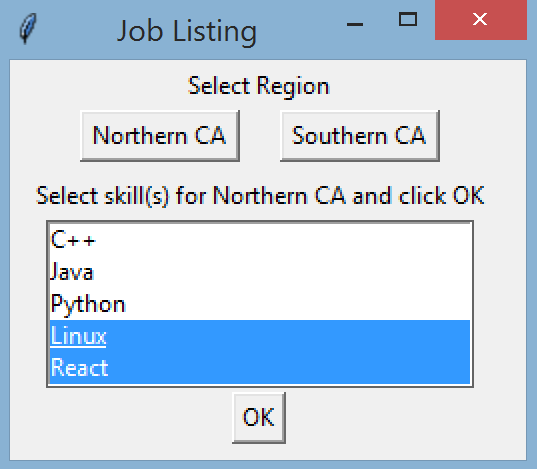
The lab4process.py is very similar to the lab4thread.py file, so you should be able to re-use much of the code.

It also has 2 GUI classes: a main window class and a display window class, and both windows have the same behavior.

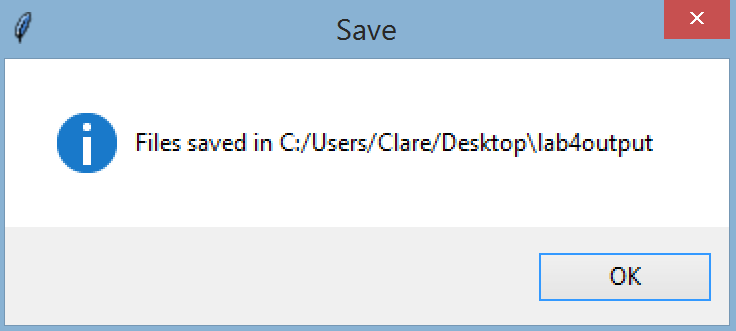
The difference in lab4process.py:

1. Change all multithreading code to multiprocessing code: instead of a maximum of 3 threads to fetch data, there are a maximum of 3 processes to fetch data.
2. For the main window, the label at the bottom doesn't display the status any more. This is because we want as little communication between the processes as possible. The bottom label now only displays "Fetching data for your N states..." where N is the number of states that the user selects.  
   This status message appears *as soon as the user clicks the OK button*. Don't wait until all the data has been fetched.  
    
3. The function that represent the task for the process should be removed from the main window class and become a global function so that the process can run it.

When done, submit lab4thread.py and lab4process.py



* The user can select one or more skills and then clicks OK to commit the selection.   
  In the example above, the user clicked the Northen CA button and then the last 2 items in the listbox.
* If the user doesn't make any selection in the listbox and clicks OK, pop up an error message to ask the user to make a selection.
* If there are user selections in the lisbox when the OK is clicked, then a file dialog window pops up to ask the user to choose a directory. The default directory that shows up at first is the user's current directory.
* When the data is retrieved from the API and stored in a file (see part 3), pop up an acknowledgment with the user's chosen directory name.



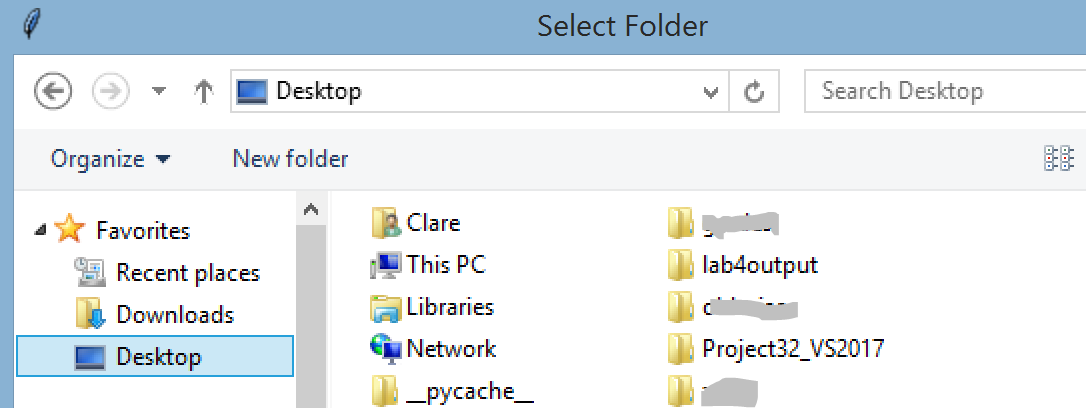
* The user can then continue to select a location or a skill from the main window.

Part 3: System

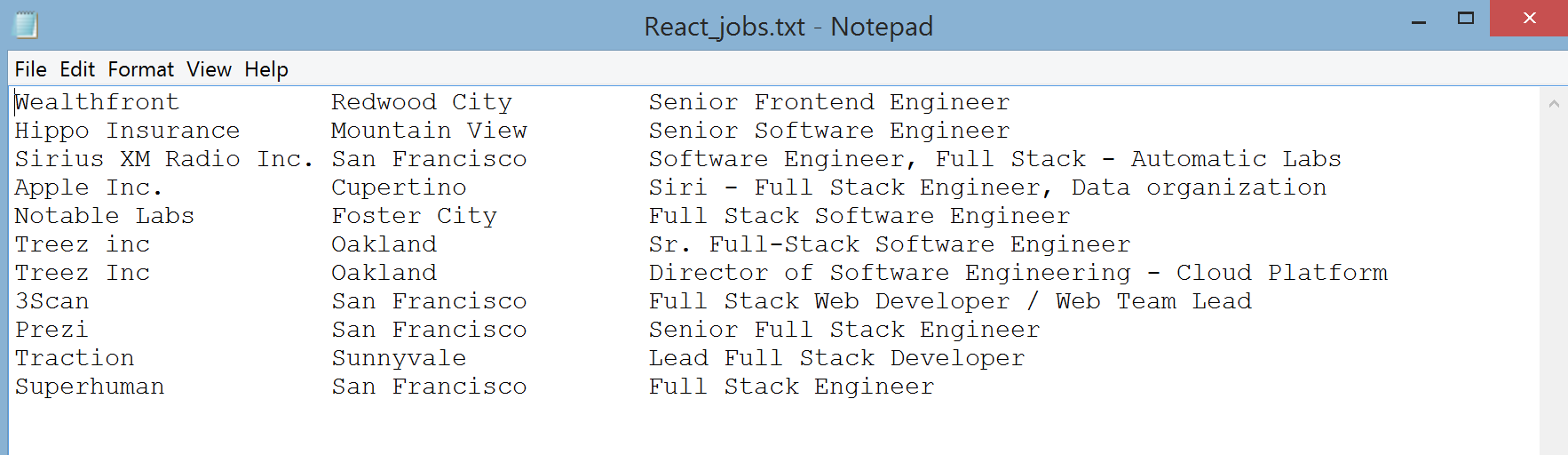
For each selected skill in the listbox, follow these steps to retrieve and store the data:

* Follow the steps of Part 1 to request data from the API with the given location and skill
* With the JSON data received, extract the company name, location, and job title for each matching job posting.  
  The location should contain the city name only. Example: "Santa Monica" and not "Santa Monica, CA"
* Pop up a file dialog window to let the user choose a directory to save the extracted data
* Create a subdirectory under the user's chosen directory, the name of the subdirectory is "lab4output". If a "lab4output" directory already exists in the user's chosen directory, then don't create it again.

Example of the file dialog window after the user selected "Desktop" as the save directory, and after the user has saved data from a previous run, so a "lab4output" directory already exists   
(Some of the directory names are greyed out for display purpose)



* If the user clicks "Cancel" or clicks X to close the file dialog window, the application is back at the main window.
* If the user selets a directory, then store the extracted information in a new text file in the "lab4output" directory, which is under the user's chosen directory.
* Name the file with the skill name, followed by "\_jobs.txt". (Make sure there's no space in the filename)  
  Example of the .txt file as shown in notepad. This file is for Northern CA location and React skill.



* Example of file creations:  
  1. If the user runs the application for the first time and chooses C++ and Linux as the skills, then in the user's chosen directory, there should be a lab4output directory, and in the lab4output directory are 2 files: C++\_jobs.txt and Linux\_jobs.txt  
  2. If the user continues at the main window and chooses Python and C++ as the skills, and then chooses the same save directory, then in the lab4output directory will be 1 additional file: Python\_jobs.txt. The new C++\_jobs.txt will overwrite the old C++\_jobs.txt
* Make sure your code is platform independent: don't use any OS-specific command or syntax.

Part 4: multithreading and multiprocessing

Step A: work with threads

* To speed up the data retrieval from Github, create as many threads as the number of skills that the user chooses from the listbox.
  + The threads need to run at the same time and independently from each other. Use a loop to start the threads so they practically all start at the same time.
  + Each thread does the task shown in Part 3 but with its own 'skill' parameter.
* Set a timer to run *right before* all the threads start, and end the timer *immediately after* the last thread is done. Print to console the elapsed time.

Step B: work with processes

* Copy the code you have into a new file called lab4process.py. Remove the code with threads and substitute in the code for processes. If you organized your lab4thread.py wisely, it should be easy to swap out threads for processes.
* Create as many processes as there are the number of skills that the user chooses from the listbox. The processes need to run in parallel, similar to the threads.
* Set a timer to run *right before* all the processes start, and end the timer *immediately after* the last process is done. Print to console the elapsed time.
* Because of the nature of multiprocessing, the code that each process will run (part 3) will need to be in a global function, and not a method of any class.
* Make sure only one main function will run when processes are created.

Step C: compare multithreading and multiprocessing for this application

* At the end of the lab4process.py file, in a comment block, compare and explain the elapsed time of threads vs processes when the user selects the same location and skills.  
  Example format: To fetch and process 3 data sets, threads took longer (or shorter or the same time) than processes because threads run on the same CPU core (or processes can truly run in parallel or ...)

Extra credit (2pts)

* Choose the same 3 data sets to download.
* Run lab4thread.py with 2 threads, 3 threads, 4 threads, 5 threads, 6 threads and record each elapsed time
* Run lab4process.py with 2 processes, 3 processes, 4 processes, 5 processes, 6 processes and record each elapsed time
* Plot the number of threads/processes vs elapsed time (on the same plot)
* Upload to the Lab 4 extra credit forum to share with the class:
  + The elapsed times for threads and for processes
  + The plot
  + The CPU count of your system . Use: print(psutil.cpu\_count()) after import psutil  
    Documentation: <https://psutil.readthedocs.io/en/latest/>

When done, turn in 2 files: lab4thread.py and lab4process.py