Lab 2 – Lists

(a.k.a. Let's Start Working with Lists)

Release: 31 Aug 2020 (Mon, Week 4)
2 weeks to attempt
Due: 13 Sep 2020, 11pm (Sun)

Some Words:

There are two objectives to this lab:

(i) try writing some Python code that uses 1D and 2D lists, and
(ii) compare the performance (speed) of two different algorithms that have different big-O time complexities for large data sets.

Searching and sorting algorithms will only be covered in class during week 4. You are strongly encouraged to prepare for this lab exercise and for week 4's class by reading chapters 4 and 5 of J.S. Conery (specifically "section 4.2 – Implementing Linear Search" (p.96; PDF p.110), "section 5.1 – The Binary Search Algorithm" (p.120; PDF: p.134) and "section 5.2 – Implementing Binary Search" (p.122; PDF: p.136).

Instructions:

- There are 2 questions in this exercise to be completed individually.
- For this exercise, your team ID is your name (i.e. you are the only member in your team).
- You need to submit code for this exercise at the Submission Server. No written submission is required.
- Edit **lab2a.py** and **lab2b.py** that are given to you, and submit them to the Submission Server.
- You can submit your solutions to the Submission Server as many times as you wish, but the final submission on the deadline will be taken as your final submission.
- Before starting this lab, do go through the appendices behind. Specifically, Appendix B gives you a good idea of how you can manipulate lists in Python. Use IDLE to try these examples.

Lab 2a

Lab 2a is an exercise involving 1-dimensional (1D) lists. You are given a fully functional algorithm that performs linear (sequential) search, and are required to improve the algorithm to reduce the time taken to perform the same task.

You are given a list of 1 million employee IDs in a CSV file. Here are some special characteristics of the list:

(i) The employee IDs are sorted in ascending order in the CSV file.

(ii) Employee IDs may not be consecutive. i.e. it is possible that employee IDs 5, 7, 8 exist, but there could be no ID 6. (This happens because employees leave the company, and IDs are not recycled.)

In Python, this employee list can be represented as a list of integers. The first employee ID read from the CSV file will be the first element in the list, the second employee ID will be the second element in the list, and so on.

You are given the following files for this exercise:

File name	Description	Comments
lab2a.py	Contains the **perform_once** and **exist** functions that you will write.	You need to modify and submit this file. This is the only file that you may submit. Do not modify the file name or the function signatures in this file.
lab2a_main.py	Loads **lab2a** and calls the **perform_once** function one time, followed by the **exist** function 500,000 times with random argument values.	Do not submit this file; use it to check the correctness and length of time your **exist** function takes before submitting it to the Submission Server.
data folder containing CSV files	Contains **employees_1mil.csv,** and **employees_100.csv**. These are data files read by line 12 of **lab2a_main**.	Do not submit these files. Each row in these data files represents an employee, and contains an employee ID. The employee IDs are sorted in ascending order in this CSV file. **employees_1mil.csv** contains 1 million rows, and **employees_100.csv** contains only 100 rows. You may edit the **DATA_FILE_NAME** constant on line 12 of **lab2a_main** to use **employees_100.csv** for smaller-scale testing purposes.

Notes:

- Study **lab2a.py** and **lab2a_main.py** to get a rough idea of what they do.
- **lab2a.py** contains 2 functions: **perform_once** and **exist**.
- You are given a function called **exist** that takes in 2 parameters:
  - **id** (an integer representing the employee ID)
  - **employee_list** (a list of employee IDs)

  This function checks if **id** is an existing employee ID on the **employee_list** read from the data file and returns **True** (if so) or **False** (if not).
- You don't have to bother about how to read the CSV file. When **lab2a_main** runs, all the employee IDs would be read and put into a list called **employee_list**, which your functions can use.
- You are allowed to use the **employee_list** variable in **perform_once** and **exist** freely.
- You are allowed to modify **perform_once** if desired. **perform_once** will be called one time, followed by multiple calls to **exist** (with random arguments) in **lab2a_main.py**. You do NOT need to modify **perform_once** if you don't want to.

- Lists in Python have an **in** keyword to check if an item is in the list:

  e.g.

```
>>> a = [5, 7, 2, 4]
>>> 7 in a
True
>>> 8 in a
False
```

  Using the **in** keyword to solve this problem does not help much because the **in** keyword uses linear search to perform the check.

Your task:

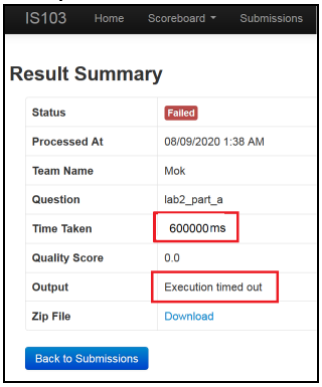
- **exist** is a fully functional function, but it uses linear search to determine if **id** exists. The code in **lab2a_main.py** calls **exist** 500,000 times to search a list of employee IDs. If you use the list with 1 million IDs (**employees_1mil.csv**), it takes approximately 13 hours to complete running on a modern laptop.
- Your challenge is to come up with a significantly faster algorithm than linear search and replace the code in **exist** with your new algorithm.
- To ensure that your **exist** function returns the correct value, the last few lines of code in **lab2a_main.py** checks the first 100 results (of your 500,000 function calls) for correctness. (The assumption here is that if the first 100 results are correct, all your results are correct.)

To submit:

- **lab2a.py** (at submission server). Edit the comments at the top of your Python file to indicate your name and section.

Assessment:

- This exercise is not graded but submission of a working answer is mandatory.
- The "Quality Score" is irrelevant for this exercise, and will always be set to 1.0 if your solution works correctly.
- Your new algorithm should strive to achieve a low "Time Taken" value. Your submitted code must complete running (i.e. your function should return a value when invoked) **within 10 mins** on the server. If your **exist** function takes more than 10 minutes to return, your submission will be marked as "Failed". The Submission Server has been deliberately constrained so that it performs slower than a modern personal laptop, so it is important that you test your submission on the Server in order to determine if your solution meets this time constraint. This is what you will see if it takes more than 10 minutes (600,000 ms) for your function to run:



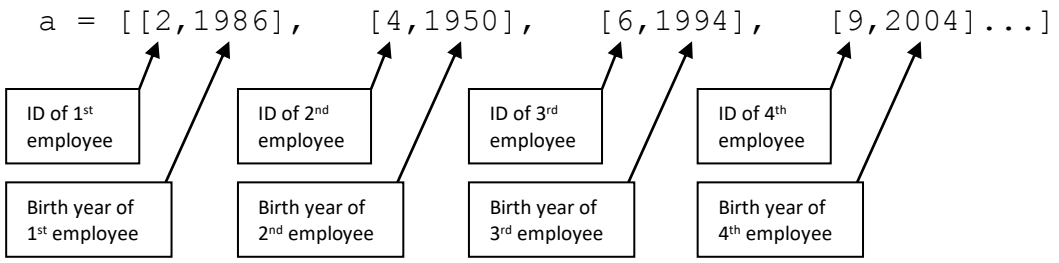
Lab 2b

Lab 2b is an exercise to get you familiar with 2-dimensional (2D) lists.

You are given a list of 1 million employee IDs and their respective birth years in a CSV file. Like the previous question, all employee IDs are sorted in ascending order in the CSV file, and all employee IDs may not be consecutive. The only difference here is that the birth year of each employee is included in each row. Here is an extract of how the CSV file may look like:

```
2,1986
4,1950
6,1994
9,2004
12,1988
…
```

In Python, this employee list can be represented as a 2D list which looks like this:



You are given the following files for this exercise:

File name	Description	Comments
lab2b.py	Contains the **perform_once** and **get_IDs_with_birth_year** functions that you will write.	You need to modify and submit this file. This is the only file that you may submit. Do not modify the file name or the function headers in this file.
lab2b_main.py	Loads **lab2b** and calls the **perform_once** function one time, followed by the **get_IDs_with_birth_year** function 100 times with random argument values.	Do not submit this file; use it to check the correctness and length of time your **get_IDs_with_birth_year** function takes before submitting it to the Submission Server.
data folder containing CSV files	Contains **employees_birthyear_1mil.csv**, **employees_birthyear_200k.csv** and **employees_birthyear_100.csv**. These are data files read at line 20 of **lab2b_main**.	Do not submit these files. Each row in these data files represents an employee, and contains an employee ID and his birth year. The 3 given CSV files contain 1 million, 200000 and 100 rows respectively. You may edit the **DATA_FILE_NAME** constant on line 12 of **lab2b_main** to use

		employees_birthyear_100.csv for smaller-scale testing purposes. (However, the test cases for correctness testing used **lab2b_main** are only applicable for **employees_birthyear_1mil.csv**).

Notes:

- Study **lab2b.py** and **lab2b_main.py** to get a rough idea of what they do.
- **lab2b.py** contains 2 functions: **perform_once** and **get_IDs_with_birth_year**. **get_IDs_with_birth_year** takes in 2 arguments (**year** and **employee_with_birthyear_list**).
- You don't have to bother about how to read the CSV file. When **lab2b_main** runs, all the employee IDs and birth years would be read and put into a 2D list called **employee_with_birthyear_list**, which your functions can use.
- You are allowed to use the **employee_with_birthyear_list** variable in **perform_once** and **get_IDs_with_birth_year** freely.
- You are allowed to modify the code into **perform_once** if desired. **perform_once** will be called one time, followed by multiple calls to **get_IDs_with_birth_year** (with random arguments) in **lab2b_main.py**. You do not have to modify **perform_once** if you don't want to.

Your task:

- For now, **get_IDs_with_birth_year** always returns an empty list. You are required to edit the code in **get_IDs_with_birth_year** so that it returns a list of IDs (as integers) with matching birth years as the argument (**year**). If there is no matching employee, this function returns an empty list (i.e. **[]**).

- Examples:
  Test case #1: **get_IDs_with_birth_year(1949)** should return the following list:
  ```
  [101, 201, 1999632, 1999649]
  ```
  because in **employees_birthyear_1mil.csv**, there are only 4 employees born in 1949, and their IDs are 101, 201, 1999632 and 1999649.

  Test case #3: **get_IDs_with_birth_year(2015)** should return an empty list:
  ```
  []
  ```
  because in **employees_birthyear_1mil.csv**, there is no employee born in 2015.

  There are 5 test cases which check if your **get_IDs_with_birth_year** function returns the correct list in **lab2b_main.py**.

- The code in **lab2b_main.py** calls **get_IDs_with_birth_year** 100 times to search a list of 1 million employees with random arguments between 1950 and 2005.

To submit:

- **lab2b.py** (at submission server). Edit the comments at the top of your Python file to indicate your name and section.

Assessment:

- as for **2(a).**

Appendix A: Additional Resource – Binary Search Algorithm

Binary search will be covered in week 4, but you can read Chapter 5 (Divide and Conquer) of JSC to find out more about binary search for this lab.

The Python implementation of binary search provided in JSC is given here:

```
# adapted from p. 124 (PDF p.138) of JSC
# from PythonLabs.RecursionLab import print_bsearch_brackets

# arguments:
#   - a: sorted list containing a list of integers
#   - x: key (integer to search for)
# returns:
#   - index, (position in a) if x is found in a
#   - None, if x cannot be found in a

def bsearch(a, x):
  "Use binary search to find x in list a"
  lower = -1
  upper = len(a)

  while upper > lower + 1:
    mid = (lower + upper) // 2
#   print_bsearch_brackets(a, lower, mid, upper)

    if a[mid] == x:
      return mid     # found it!! Return index

    if x < a[mid]:
      upper = mid    # discard right side
    else:
      lower = mid    # discard left side

  return None  # completed while loop; no match found
```

Appendix B: Some Ways to Manipulate Lists in Python

You are encouraged to read every section in this appendix and try the statements in IDLE.

Retrieving Elements from a 1D and 2D List

A 2D list is simply a list of lists, as the examples below show.

```
>>> a = [4, 6, 2, 1, 8, 0]
>>> len(a)    returns the number of elements in the list
6

>>> b = [[4, 6], [2, 1, 8], [0, 1], [], [1]]
>>> len(b)    returns the number of elements in the 1st-D list (There are 5 elements in the 1st list – [4,6] is the first element. [2,1,8] is the 2nd element. [0,1] is the 3rd element. [] is the 4th element and [1] is the 5th element.
5
```

In order to access an element in a list, you can do this:

```
>>> a[0]              returns the 1st element in a (index 0)
4
>>> a[1]              returns the 2nd  element in a
6
>>> a[2]              returns the 3rd element in a
2
>>> a[len(a)-1]       returns the last element in a, because (len(a)-1) is the last index of a. You can also use a[-1]
0
>>> a[-1]             same as a[len(a)-1]
0
>>> a[len(a)-2]       returns the 2nd last element in a
8
```

If you try to retrieve something that doesn't exist, you will get an IndexError. In this case, a[len(a)] doesn't exist because the last element in a is a[len(a) -1], thanks to the fact that indexing starts from 0 instead of 1 for lists:

```
>>> a[len(a)]
IndexError: list index out of range
```

The same ideas can be applied on 2D lists. Let's study some examples with b:

```
>>> b[0]              returns the 1st element in b. Note that the 1st element of b is itself a list
[4, 6]
>>> b[1]              returns the 2nd  element in b. Which is a list as well.
[2, 1, 8]
>>> b[2]              returns the 3rd element in b.
[0, 1]
>>> b[len(b)-1]       returns the last element in b
[1]
>>>b[-1]              same as b[len(b)-1]
[1]
>>> b[len(b)-2]       returns the 2nd last element in b
[]
```

So, how do you access an element in the 2D list? For example, the 1st element of b is a list with 2 integers (4 and 6):

```
>>> b[0]              returns the 1st element in b.
[4, 6]
```

In order to retrieve the value 4, you can do this:

```
>>> b[0][0]           returns the 1st element of the 1st element in b.
4
```

In order to retrieve the value 6, you can do this:

```
>>> b[0][1]           returns the 2nd element of the 1st element in b.
```

```
6
```
As expected, if you try to retrieve something that doesn't exist, you will get an IndexError. As in this case:
```
>>> b[0][2]          returns the 3rd element of the 1st element in b (which does not exist).
IndexError: list index out of range
```

Inserting Elements into a List

Study the examples below
```
>>> c = []           c is an empty list. Currently there's nothing there and len(c) will return 0.
>>> len(c)           returns the number of elements in the list
0
```
Use append() to insert elements into the list:
```
>>> c.append(9)
[9]
>>> len(c)           now there is 1 element in c. And its length is 1.
1
>>> c                Let's confirm the values in the list
[9]
>>> c.append(88)     Let's insert another element into the list. New elements will be inserted at the end
>>> c
[9, 88]
> len(c)            now the length of c is 2
2
>>> c.append(-1)     Let's insert a few more elements
>>> c.append(6)
>>> c.append(0)
>>> c
[9, 88, -1, 6, 0]
```

What about 2D lists? Check this out:
```
>>> d = []                  d is an empty list.
>>> d.append([8, 9, 7])     Insert a list ([8,9,7]) into d
>>> d                       Note that there are 2 pairs of square brackets!! [8, 9, 7] is a list, and it's
[[8, 9, 7]]                 considered 1 element in the 1st-level list.
>>> len(d)                  Note that len(d) is still 1. There is only 1 element in d, which is the 1st-level list.
1
>>> len(d[0])    On the other hand, d[0] refers to the 1st element in d, which is a list. So, len(d[0]) returns 3
3
>>> d.append([1, 2, 3])          Let's insert a few more elements into d for the fun of it.
>>> d.append([])
>>> d.append([2])
>>> d
[[8, 9, 7], [1, 2, 3], [], [2]]
```

Deleting Elements from a List

Study the examples below:
```
>>> e = [9, 88, -1, 6, 0, 88]
```
You can remove elements from the front as well using the pop function like this:
```
>>> e.pop(0)         returns the 1st element from the front and deletes it from the list
9                    if you need the popped out value, store it in a variable like this: temp = e.pop(0)
>>> e          if you check the values in e again after pop, notice that it has one fewer element now
[88, -1, 6, 0, 88]
```
You can remove elements from the middle by specifying the index using pop like this:
```
>>> e.pop(2)         returns and deletes the 3rd element (index =2)
6
>>> e
```

```
[88, -1, 0, 88]
```
Instead of removing an element based on its index, you can remove elements that match a certain criteria like this:
```
>>> e.remove(88)    deletes the first element in the list that is 88
>>> e
[-1, 0, 88]         notice that only the first occurrence of 88 is deleted, not both
>>> e.remove(1)     Python throws an error if there is no such element in the list
Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list [-1, 0]
```

Therefore, it is a good idea to check if an element is in the list before removing it
```
>>> if 1 in e:
...     e.remove(1)
...
>>> e
[-1, 0, 88]
```

Sorting Elements in a 1D List

Use the sort() or sorted() functions to sort a list.
```
>>> f = [9, 88, -1, 9, 0, 88]
```
sorted() returns a new list, but doesn't change f.
```
>>> temp = sorted(f)      the sorted() function returns an entirely new list that is sorted
[-1, 0, 9, 9, 88, 88]
>>> temp                  temp now stores a new and independent list
[-1, 0, 9, 9, 88, 88]
>>> f                     remember that the original list (stored in f) is unchanged
[9, 88, -1, 9, 0, 88]
```

Unlike the sorted() function, calling a list's sort() function actually modifies f, so that f is now permanently sorted.
```
>>> f.sort()              list.sort(). As opposed to sorted(list)
>>> f
[-1, 0, 9, 9, 88, 88]
```

Here's how you can sort in descending order (biggest to smallest)
```
>>> f.sort(reverse=True)
>>> f
[88, 88, 9, 9, 0, -1]
```
Try sorted(f, reverse=True) as well.

Sorting Elements in a 2D List

Here's an example of sorting a 2D list. The 1st element in the "inner list" (highlighted in yellow) will be used for sorting by default.
```
>>> g = [[8, 9, 7], [1, 2, 3], [2, 4, 6], [0]]
>>> g
[[8, 9, 7], [1, 2, 3], [2, 4, 6], [0]]
>>> g.sort()
>>> g
[[0], [1, 2, 3], [2, 4, 6], [8, 9, 7]]
```

You can sort by the 2nd element of the "inner list" as well (highlighted in green):
```
>>> h=[[1, 4, 6], [2, -1, 0], [8, 5, 7], [3, -1, -1]]
>>> h.sort(key=lambda x: x[1])       # x[1] means the 2nd element
>>> h
[[2, -1, 0], [3, -1, -1], [1, 4, 6], [8, 5, 7]]
```

Sorting by the 3rd element of the "inner list" is easy as well:

```
>>> h=[[1, 4, 6], [2, -1, 0], [8, 5, 7], [3, -1, -1]]
>>> h
[[1, 4, 6], [2, -1, 0], [8, 5, 7], [3, -1, -1]]
>>> h.sort(key=lambda x: x[2])       # x[2] means the 3rd element
>>> h
[[3, -1, -1], [2, -1, 0], [1, 4, 6], [8, 5, 7]]
```

You can sort by 2 criteria. E.g. you want to sort by the 2nd element, then the 1st element (in this order). i.e. if there is a tie for the 2nd element, the 1st element's value is used for sorting.

```
>>> h=[[1, 4, 6], [2, -1, 0], [8, 5, 7], [3, -1, -1]]
>>> h
[[1, 4, 6], [2, -1, 0], [8, 5, 7], [3, -1, -1]]
>>> h.sort(key = lambda x: (x[1], x[0]))      2nd element (x[1]), then 1st element (x[0])
>>> h
[[2, -1, 0], [3, -1, -1], [1, 4, 6], [8, 5, 7]]
```

In fact, you can sort by almost anything. Here is an example of sorting by length (i.e. the number of elements) of the "inner list" in a 2D list

```
>>> i = [[3, 4, 1], [2], [7, 8, 9, 1, 5], [1, 2], []]
>>> i
[[3, 4, 1], [2], [7, 8, 9, 1, 5], [1, 2], []]
>>> i.sort(key = lambda x: len(x))
>>> i
[[], [2], [1, 2], [3, 4, 1], [7, 8, 9, 1, 5]]
```

Sort by descending (biggest to smallest) order of length:

```
>>> i.sort(key = lambda x: len(x), reverse = True)
>>> i
[[7, 8, 9, 1, 5], [3, 4, 1], [1, 2], [2], []]
```

Miscellaneous

Removing repeated elements in a list

```
>>> j = [9, 88, -1, 9, 0, 88]
>>> j
[9, 88, -1, 9, 0, 88]
>>> list(set(j))   only unique elements are returned. You can only find one 9 and one 88 in the returned list
[9, 88, -1, 0]
>>> j                          however, notice that this function does NOT change the value of j
[9, 88, -1, 9, 0, 88]          j still contains the original elements
>>> j = list(set(j))           you could change the value of j by assigning the result of this function
>>> j
[88, 9, -1, 0]
```

Common Errors when Working with Lists

Consider the following:

```
>>> a = [77,88,99]
>>> a
[77, 88, 99]
>>> a[0]      retrieve the first element in a
77
>>> a[1]      retrieve the 2nd element in a
88
>>> a[2]      retrieve the 3rd element in a
99
>>> a[3]      retrieve the 4th element in a. Causes an IndexError because the list does not have that index
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> a = None       Try this!!!
>>> a[0]           You will see the following error:
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'NoneType' object is not subscriptable
```

This error means that you cannot apply [] on a "None". You can only use the square bracket operator on a variable that stores either a (i) string, or (ii) a list. See these examples:

```
>>> s = "apple"
>>> s[0]    # retrieve the first character in a string
'a'
>>> s = ["apple", "orange"]
>>> s[0]    # retrieve the first element in a list
'apple'

>>> s = 10
>>> s[0]    # does not make sense. Causes an error
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    s[0]
TypeError: 'int' object is not subscriptable

>>> s = None
>>> s[0]    # does not make sense. Causes an error
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    s[0]
TypeError: 'NoneType' object is not subscriptable
```

END